AFIT/GCS/ENG/99M-14

A Structured Approach to Software Tool Integration

THESIS

Penelope Ann Noe
1Lt, USAF

AFIT/GCS/ENG/99M-14

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1999	Master's Thesis

4. TITLE AND SUBTITLE
A Structured Approach to Software Tool Integration

5. FUNDING NUMBERS

6. AUTHOR(S)
Penelope A. Noe

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB, OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/99M-14

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Roy F. Stratton, AFRL/IFTD
525 Brooks Rd.
Rome, NY 13441-4505
(330) 315-3004 (DSN 587-3004)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Dr. Thomas C. Hartrum
Thomas.Hartrum@afit.af.mil
(937)255-3636x4581

12a. DISTRIBUTION AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

As the trend towards commercial off-the-shelf (COTS) software continues, civilian companies and government agencies alike are battling with the challenge of making multiple software packages and applications work together. Many of these companies and agencies have attempted to integrate the software tools to form a coherent system that satisfies their goals. often without the use of a step by step approach guiding the effort. Many researchers in the field of software tool integration have determined the areas that need to be addressed when tools are integrated. Some researchers have developed and expanded upon a theoretical model for integration. This model of tool integration aids in understanding what types of integration need to be performed, but does not provide a set of steps to aid in completing the integration. The methodology developed as part of this thesis research is based upon this model of integration. It provides a method of characterizing the tools being integrated and offers guidance on how to integrate them in a step by step manner.

A software development tool, AFITtool, has been developed at the Air Force Institute of Technology (AFIT) to build software based on a formal requirements specification. The process of developing executable code from a requirements specification is based on mathematically provable, correctness-preserving transformations. Researchers at AFIT realized that some of AFITtool's shortcomings could be addressed by taking advantage of the capabilities of other tools. As part of this research, three tools were chosen to integrate with AFITtool and performing the integrations served to demonstrate the effectiveness of the methodology developed, while addressing specific shortcomings of AFITtool.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software tool integration, tool integration methodology, integrated software environment.	117
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

AFIT/GCS/ENG/99M-14

A Structured Approach to Software Tool Integration

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Systems

Penelope Ann Noe, B.A. Computer Science

1Lt, USAF

March, 1999

AFIT/GCS/ENG/99M-14

A Structured Approach to Software Tool Integration

Penelope Ann Noe, B.A. Computer Science

1Lt, USAF

Approved:

Thomas C. Hartrum — 4 Mar 1999

Dr. Thomas C. Hartrum — Date
Committee Chair

Robert P. Graham Jr — 4 Mar 1999

Maj. Robert P. Graham, Jr. — Date
Committee Member

Michael Talbert — 4 Mar 1999

Maj. Michael L. Talbert — Date
Committee Member

Acknowledgements

Over the past 18 months, many people have been called upon to provide much needed support. To those people, I say "Thank You." This thesis work could never have been accomplished without the support of my family and friends. I can never thank them enough for all of their love and encouragement. While there are too many to name, I hope they all realize how much I appreciate them. I would also like to thank my advisor, Dr. Hartrum, for always being available for encouragement and guidance. He helped me to develop something to make us both proud. He was always pushing for a little more, to make it better, stronger, and clearer. My other committee members, Maj Graham and Maj Talbert were also very helpful with all of their fresh ideas and thought-provoking questions. Finally, I would like to thank my wonderful office mates, Capt Dave VanVeldhuizen and Maj Tom Schorsch. They were always there when I needed them, whether it was to offer words of encouragement or to make me laugh. All in all, this has been an experience I will not soon forget, nor do I wish to.

Penelope Ann Noe

Table of Contents

List of Figures

List of Tables

Abstract

As the trend towards commercial off-the-shelf (COTS) software continues, civilian companies and government agencies alike are battling with the challenge of making multiple software packages and applications work together. Many of these companies and agencies have attempted to integrate the software tools to form a coherent system that satisfies their goals. However, most of this integration work was accomplished without the use of a step by step approach guiding the effort.

Many researchers in the field of software tool integration have determined the areas that need to be addressed when tools are integrated. Some researchers have developed and expanded upon a theoretical model for integration. This model of tool integration aids in understanding what types of integration need to be performed, but does not provide a set of steps to aid in completing the integration. The methodology developed as part of this thesis research is based upon this model of integration. It provides a method of characterizing the tools being integrated and offers guidance on how to integrate them in a step by step manner.

A software development tool, *AFIT*tool, has been developed at the Air Force Institute of Technology (AFIT) to build software based on a formal requirements specification. The process of developing executable code from a requirements specification is based on mathematically provable, correctness-preserving transformations. Researchers at AFIT realized that some of *AFIT*tool's shortcomings could be addressed by taking advantage of the capabilities of other tools. As part of this research, three tools were chosen to integrate with *AFIT*tool and performing the integrations served to demonstrate the effectiveness of the methodology developed, while addressing specific shortcomings of *AFIT*tool.

A Structured Approach to Software Tool Integration

I. Introduction

Commercial off-the-shelf (COTS) software is more widely used now than ever before. As COTS becomes more capable of handling larger and larger tasks, more companies rely on it to assist in achieving their daily goals. Besides using COTS, the government also creates software for use by other government agencies, usually termed government off-the-shelf (GOTS) software. As the number of COTS and GOTS packages used by each company or government agency increases, so does the need for them to work together.

While each company or agency has a mission that drives its daily activities, often there is not just one software tool capable of accomplishing the mission. Each agency usually uses several tools in combination to accomplish their mission. One drawback of purchasing off-the-shelf software is the necessity of accepting tools as they are rather than using custom-built tools. In some cases, companies are turning to tool integration to form a cohesive environment that is similar to a custom-built tool.

The Air Force Institute of Technology (AFIT) has not missed this trend. Research conducted at AFIT to create a formal method of software development, from requirements specification to code generation, has produced *AFIT*tool. Although the tool has been developed in-house, it is lacking in some functionality that is offered by COTS tools. In order to take advantage of the capabilities of the COTS tools, one or more of them could be integrated with *AFIT*tool. At the start of this research, a structured method of tool integration was not in use at AFIT. It was recognized that such a method would be helpful in integrating *AFIT*tool with the desired COTS tools. Developing a methodology for use in software tool integration became the focus of this research, using *AFIT*tool to demonstrate the validity of the methodology.

The rest of this chapter describes the motivation for integrating *AFIT*tool with other tools, as well as the motivation for developing a structured approach to tool integration.

The first section gives further information on the background of *AFIT*tool and tool integration in general. The next section describes the problems encountered in tool integration and *AFIT*tool, followed by a discussion of the past efforts in tool integration and the development of *AFIT*tool. The last three sections of the chapter describe the scope, approach and assumptions of this research effort.

1.1 Background

The goal of using several tools in harmony to achieve an organization's mission can often be realized by integrating the necessary tools. In the past, researchers have developed models of integration to give structure to tool integration efforts. One model proposes using an integration tool as the framework, fitting the tools into this framework [9]. Other models are built on the premise that there are levels of integration that must be addressed in any integration effort.

At the same time, the software engineering industry has been striving to develop a formal, consistent method for generating provably correct software. Part of this effort has included using formal methods to formulate a requirements specification that can then be used to produce executable code (see Figure 1). *AFIT*tool has been developed to satisfy this need. It uses an internal abstract syntax tree (AST) for each object class specified. Specifications, written as *Z* (zed) schemas using LaTeX syntax, are converted and parsed into these ASTs.

One of the essential aspects of software engineering is code reuse. Much effort is put forth to specify, develop, and store code that can be reused. Often this creates more work for the developers and does not reduce future effort. In order to reuse code, it has to be rather generic, requiring the reuse developers to tailor the code for their purpose. Tailoring is often as much work as developing from scratch. However, many researchers claim that efforts to exploit reuse of domain knowledge can be very successful [5]. The knowledge gathered in the software development process is often generic enough that it can be used in other applications. For example, if an application is developed to simulate a cruise missile, much of the domain knowledge gathered in the development process could be reused in an application to simulate a rocket. Although the domains are different, they

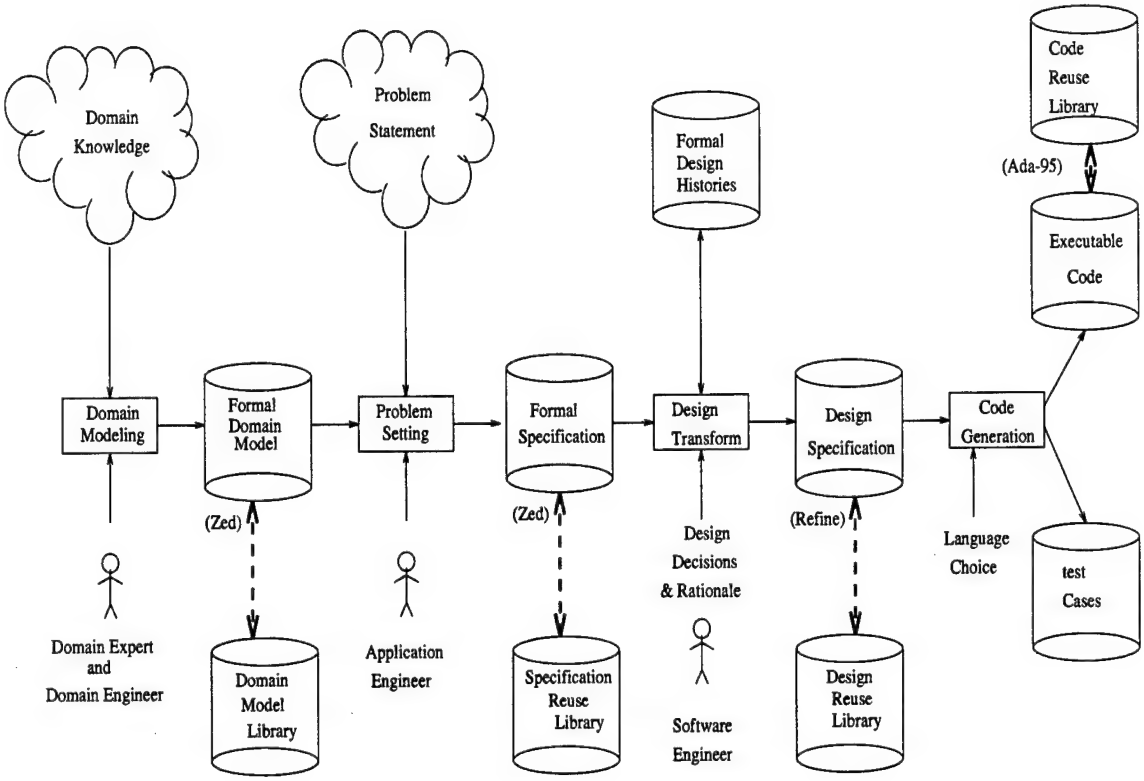


Figure 1 Overall Transformation System Concept

have many similarities because they are both composed of similar components, such as a jet engine, fuel tank and throttle.

In addition, the formal language used in the *Z* specification is often difficult for the customer to understand. Although the specification is mathematically provable, and therefore easier to extend to code generation, most customers are not familiar with formal methods, nor are they educated in them. Therefore, the customer may approve of an incorrect specification due to lack of understanding, defeating the primary goal of formal methods: to create the correct system. Additionally, formal methods are difficult for most developers, managers, and other team members to understand.

By integrating *AFIT*tool with existing tools that handle the previously mentioned areas, it is possible to develop an integrated toolset that provides complete support for generating provably correct code. Adding to *AFIT*tool would not affect its strength, the use of formal methods to develop correct software, but would make it a more robust tool, possibly offering domain model reuse and other methods of inputting domain models.

1.2 Problem

Since there is not an industry-wide standard for tool integration, many tool integration efforts have been haphazard at best. Often there has not been a clear plan on how to integrate the tools, causing the integration efforts to miss the goal of a seamlessly integrated system. Although the system may still work and achieve the mission of the company, it may not be easy to use and it may cause confusion on the part of the user.

For the most part, the developers involved in integrating tools hope the tools work together well, in such a way that users are not aware of the existence of multiple tools. If the integrated toolset seems like several tools, the users must know how to work with several different tools, instead of just learning one tool. Not only can this cause confusion, but it can also cause errors that lead to user frustration. Another possible problem after tool integration is that it may restrict the set of correct inputs, forcing the users to remember new rules. In the end, they may revert to using the separate tools so they know exactly what tool they are using and what rules to follow.

At the start of this research, *AFIT*tool was a single tool which partially addressed the problem of generating executable code from a formal requirements specification. Since the executable code reflects the requirements specification, it is critical that the specification correctly represent the system as the user intends. By offering another view of the specification or adding an automated level of checking, it is possible to detect some types of errors in the specification. One way to add these additional layers of correctness checking is to integrate *AFIT*tool with existing tools that perform the desired functions.

Although the ASTs created by *AFIT*tool can be made persistent, there is not an easy mechanism in place to retrieve, search or customize the ASTs, nor is there a mechanism to navigate any domain knowledge gathered in the process of developing the ASTs. Since there are many tools that provide for persistence of data, the key issue is developing an interface between one of these tools and *AFIT*tool. If accomplished, storing the AST has great potential for future development efforts in the area of reuse. Giving the ability to use a previously stored knowledge representation allows users to extend or customize knowledge in a domain, rather than parsing the whole domain every time.

In the market today, there are several types of database management systems that provide data persistence. The current trend is to develop an Object-Oriented Database Management System (ODBMS) that captures the semantics of the data by encapsulating it into units called objects [32]. There are also more traditional methods such as flat file systems and relational databases. A middle ground is found in the area of extended relational databases and object relational databases [28].

Another consideration is integrating *AFIT*tool with existing CASE tools to create a friendlier environment for the user. A CASE tool, usually graphically based, includes a generally uncomplicated method to input, modify and output key information in the software engineering life cycle. Many allow the user to develop requirements specifications, design specifications and code. In this approach, *AFIT*tool would continue to be used to enforce the formalisms, but the CASE tool would provide a friendly interface for the user, rather than the text-based interface in existence now. Again, the key issue is to develop an interface to ensure the semantics of the model are enforced as it is transferred from a graphical form to formal language specification and vice versa [7]. There are many

large-scale CASE tools available, including Rational Rose [21] and ERwin, and several smaller-scale tools such as the Knowledge Based Software Assistant (KBSA) Advanced Development Model (ADM) [8] and Aesop [11].

The focus of this research effort is summarized in the following statement.

Problem Statement: *Show that tool integration can be accomplished by following a structured approach that can be applied to any pair of software tools. In addition, show that the appropriate tools are chosen for integration based on identified shortcomings of an existing system. Demonstrate these assertions by integrating* AFIT*tool with one or more tools, while maintaining the semantics and the formalisms of the specification.*

1.3 Initial Assessment of Past Efforts

Researchers on tool integration have developed models for integration that range from proposing the use of a tool as a framework to the use of a highly theoretical model as the basis for integration. The model most widely published was first developed by Anthony Wasserman and has been modified and extended by several researchers. Wasserman's model is directed toward computer-aided software engineering (CASE) environment development [36]. His research offers a way of examining which tools should be integrated in order to meet the software development goals. Chapter 2 describes the five integration classes of Wasserman's model in more detail. None of the proposed models, however, offer a step by step methodology to use during the actual integration. Additionally, they do not offer any guidance on the best way to approach the integration of the tools. The thrust of his research seems to be ensuring that theoretically all aspects of the integrated toolset are intact [36].

Past efforts in the research of formal methods at AFIT have produced the tool that exists today. At the start of this research, *AFIT*tool took only a formal specification as input, written in *Z* in LaTeX. The specification was then parsed into a domain abstract syntax tree (AST). The domain AST was then transformed by a series of partially implemented design transformations and a design representation AST was created [14]. The next step in the process, transforming to a coding language AST (specifically Refine Ada),

was also partially implemented [30]. The transformations from Z to Ada were completed in research efforts at AFIT concurrent with this one [23] [33].

*AFIT*tool supports writing the domain AST to a file, which can then be loaded at a later time. However, to load multiple domains required an extensive amount of manual preprocessing to put the files in the format expected by *AFIT*tool. There was also no central repository for domain ASTs, meaning each person only knows of his or her own domain ASTs. Additionally, there was no naming convention for these ASTs, so even if a central repository were created, it would be very difficult to retrieve the correct AST.

1.4 Scope

This research effort was concerned with developing a methodology for integrating software tools by first examining the characteristics of the tools involved and then following a general set of guidelines to perform the integration. Although *AFIT*tool was used as the demonstration system, this research was not primarily concerned with changing *AFIT*tool to address all of its shortcomings. The goal was to modify *AFIT*tool in order to address some of its shortcomings and to demonstrate the feasibility of a structured approach to tool integration.

1.5 Approach

To meet the proposed research objectives, the following approach was followed:

1. ***Assess the current state of the system.*** An in-depth analysis of the current state of the *AFIT*tool system was performed, including what ASTs are built and how the various ASTs are used in the system. Shortcomings of *AFIT*tool that could be met by other existing tools, such as a persistent storage manager and one or more CASE tools, were identified.
2. ***Research current state of the art of tool integration.*** Many military, academic and commercial organizations in the software engineering field are investigating toolset integration and interoperability. A literature review was conducted to avoid dupli-

cation of effort, as well as to gather any available information on potentially useful approaches.

3. *Outline the capabilities of existing tools.* Several categories of tools were potential candidates for integration, including database management systems, graphical user interface CASE tools, software architecture tools, and drawing tools that take domain information and produce diagrams. Several tools were investigated to determine the feasibility and benefits of integrating one or more of them. The integration of one or more of these tools with *AFIT*tool benefits users of *AFIT*tool by extending and supporting its capabilities. Although these tools can be used in conjunction with *AFIT*tool without being integrated, an integrated toolset that combines formal and non-formal methods is very valuable to the users of *AFIT*tool.

4. *Derive a general framework or methodology for integrating tools.* The software engineering community has been evaluating the possibility of developing integrated toolsets for many years. An integrated toolset can be obtained by developing a tool with many functions, by integrating several existing tools, or both. A methodology for integrating existing tools would be useful by the software community as a whole. This approach is more flexible since it allows several existing tools to be used together, rather than starting from the beginning and developing a toolset.

5. *Integrate tools.* After conducting the tool analysis, three tools were chosen to integrate with *AFIT*tool. The tools, daVinci, the Acme parser, and Rational Rose 98, were chosen based on their functionality and their added benefits for *AFIT*tool.

6. *Verification and validation of the integrated system.* Using an example object domain, the ability of the integrated system to capture, store, retrieve and locate the necessary information without changing the meaning of the domain was demonstrated. This step was designed to show whether or not the tool integration was successful (verification), since the goal is not only to work with a domain, but also to ensure it is the correct representation of the domain, both when it is entered and when it is manipulated. The integrated system and the original system were used by some AFIT students to offer a comparison and determine whether or not the integrated system offers any benefits (validation).

7. *Demonstration of the methodology.* The methodology was demonstrated through the integration of three tools with *AFIT*tool. By using the developed methodology to integrate tools, covering many aspects of the methodology, the demonstrations were achieved. Since it is impossible to cover all areas of a generic methodology in the time available, a representative sample was demonstrated.

1.6 Assumptions

Several assumptions were made prior to the start of this research to promote its success. First, the availability of domain experts was necessary during this research. This included the sponsor, professors, past researchers, and documentation. Second, toolsets that were chosen as integration candidates had to be available. If a toolset was unavailable or would require major modifications, it was discarded as an integration candidate. Finally, the stability of the system was very important throughout the course of the research. As previously mentioned, other research work was ongoing to enhance *AFIT*tool, and it was necessary to have a configuration control methodology in place to ensure consistency of the system.

The next chapter provides background information used in this thesis effort, including a review of the capabilities of *AFIT*tool, a short description for each tool that was considered for integration with *AFIT*tool, an overview of commonly used integration methods, and a presentation of a well-known tool integration model. A complete description of the software tool integration methodology developed during this thesis effort is presented in Chapter 3. This methodology is a structured approach to integrating software tools, designed to be used on any integration effort. Additionally, a validation of the methodology is located at the end of the chapter. Chapter 4 contains a description of the application of this methodology to the integration of *AFIT*tool with three other software tools. The last chapter includes a summary of the research presented here, as well as recommendations for future work in this area.

II. Background

This chapter includes background information needed to integrate *AFIT*tool with other tools. The first two sections are background information on *AFIT*tool itself. This description serves as an overview of the development to this point, and as a motivation to develop the tool further to address some of its shortcomings. The third section is a review of tools that were candidates for integration. First, generalized tool requirements, based on *AFIT*tool's shortcomings, are given. These requirements are followed by the criteria used in the tool evaluation process. The next section describes a widely-known tool integration model and the modified models developed by other researchers in the field. The last section of the chapter gives an analysis of several methods of integrating tools.

2.1 Background Information on AFIT*tool*

Development accomplished at AFIT to create a formal method of software development, from requirements specification to code generation, has produced *AFIT*tool. This tool was built with Software Refinery, using the REFINE language, and uses an internal abstract syntax tree (AST) for each object class developed [30]. Specifications are converted and parsed into an AST called the domain AST. The domain AST represents the domain being modeled in the system. This AST is manipulated by the Elicitor Harvestor subsystem, designed to interactively refine a correct specification from a problem requirement based on the domain model.

The refined domain AST is then transformed by a series of correctness-preserving design transformations into the design AST. The design AST is then processed by an output grammar to go from design to code. The design transformations and code generation were further developed during this thesis cycle [23] [33]. Currently, Ada code is produced, but with modifications *AFIT*tool's design AST can be redirected to any language supported by the Reasoning Systems' Software Refinery [30]. In the current release of Software Refinery, code development is supported in COBOL, Ada, C and FORTRAN [30]. The domain AST is the first step in representing the requirements of the user, by representing the

specification. This specification can be manipulated, stored, retrieved and copied for later use.

In support of reverse engineering, AFIT has also developed the Generic Imperative Model (GIM), a generic representation of code. A GIM AST is transformed through the Object Extractor to produce a Generic Object Model (GOM) AST. The GOM, produced in the AFIT reverse engineering system, was a likely candidate for the design representation AST to be used in *AFIT*tool. During this thesis cycle, however, modifications were made to the GOM and a new design tree was specified [23] [33]. Since the reverse engineering work has already been accomplished, it is feasible to interface a reverse engineering tool to *AFIT*tool, using the design tree, for object-oriented code generation.

2.2 Shortcomings of AFIT*tool*

As part of the initial work on this thesis, it was necessary to determine the shortcomings of *AFIT*tool in order to characterize the kind of improvements, via tool integration, that could be made. The investigation of the shortcomings also facilitated the process of determining the criteria for the tool search. The following list indicates the shortcomings that existed before this research effort.

- **Interface is not user friendly** The interface to *AFIT*tool was functional, but not particularly user friendly. Most software developed in the recent past has a graphical user interface (GUI). In the past few years, people have become more comfortable with this type of interface and, in most cases, prefer it. There are several ways to add a GUI to *AFIT*tool that would not involve re-writing the entire system, and which would make it more user friendly.
- **Persistent storage of domains is limited** Although *AFIT*tool has the capability, through built-in Refine functions, to offer persistent storage, it offers only a limited amount. This is a feature that can be improved by further Refine coding or the addition of a data storage tool. Additionally, some sort of lookup facility would aid the use of the persistent storage.

- **Verification is limited** Currently, the parser in *AFIT*tool checks for a file that is properly structured, but does not check the correctness of the code entered. Since *AFIT*tool is based on formal methods, there are many ways to verify correctness. This is currently only partially implemented on the level of checking for duplicate variables and constants, and types that are not used. Other possible things to check are that constants have a type, that attributes are defined over declared or predefined types, that there is at least one attribute in each invariant constraint, there are no name conflicts between a subclass and its superclasses, and that derived types are defined over existing types [14]. Verification could be performed within *AFIT*tool or through the use of a separate tool integrated with *AFIT*tool.

- **Model analysis by the user is limited** After the input of the model, some users are not capable of performing detailed analysis of the model. By either offering a tool to analyze the model or offering a graphical view of the model to the user, another level of verification would be accomplished.

- **Transformation from *Z* specifications to code is not complete** The transformation of the domain AST to a design representation AST is not complete, making it impossible to go completely from requirements specification to code. This problem is well known by everyone who uses *AFIT*tool and is being addressed in other research efforts [23] [33].

By integrating one or more tools with *AFIT*tool to operate on the requirements specification represented by the domain AST, several of the identified shortcomings could be corrected. This research effort focused on representing the specification in a graphical form which allows the user to spot some kinds of errors more easily. Although the transformation from the domain AST to the design AST was not complete at the start of this research, identifying and integrating tools that can operate on the design was still considered.

2.3 Tool Review

This section discusses candidates for integration with *AFIT*tool based on a predetermined set of criteria. The first subsections describes the general type of tool that could be

used to improve *AFIT*tool, followed by a subsection that defines the criteria upon which tool selection was based. The following subsections describe the various classes of tools that were considered, with information on specific tools. Tools that failed to meet the criteria are not discussed here.

2.3.1 Tools that could improve shortcomings. Some of the shortcomings of *AFIT*tool could be addressed by the addition of tools to the system. The following paragraphs describe what kind of tools would improve each particular shortcoming of *AFIT*tool.

There are many options for a friendlier front end, including developing a GUI application involving one or more windows and menus, perhaps with Intervista[1] [30], or Visual Basic [6]. There are also OSF/Motif toolkit, a proprietary toolkit, and X Window application builders available on the Unix platform [31]. Although there is a learning curve to using these tools, it would improve the user interface of *AFIT*tool. In addition, a tool that already uses a GUI could be integrated with *AFIT*tool and used as the front end.

A tool to store the domains that allows querying would improve the reusability of domains. This could be the built-in Refine Persistent Object Base, discussed later, or a database management system. Alternatively, a configuration management system could be implemented that allows storing keywords to represent the domain, creating a central repository for domain models.

A tool that interprets the *Z* specification (in words or in pictures) and ensures that the interpretation meets with the meaning the user was trying to convey would enable more correctness checking. By giving the user a different view of the specification, it may be possible for the user to realize errors earlier. As an added method of correctness checking, a theorem prover to ensure the correctness of the specifications and the transformation could be added to the system. It would utilize the formal methods upon which *AFIT*tool is based, although it might involve extending *AFIT*tool to "explain" to the tool what is correct.

[1]Intervista is the GUI development package standard with Software Refinery

To achieve a true integration of *AFIT*tool and the selected tools, a framework is needed. The integration should be seamless to the user, offering the idea that there is one system, rather than many components. There are several ways to do this, including using a scripting language or a commercial framework designed to integrate tools.

2.3.2 Tool Criteria. During the tool evaluation process, seven criteria were used. These criteria were developed with the shortcomings of *AFIT*tool in mind (criteria 2 and 6) as well as the feasibility of acquiring the tool (criterion 5) and the feasibility of integrating the tool, based on the given time constraints (criteria 1, 3, 4, and 7). They address the issues of integrating tools in a feasible manner, to provide an integrated environment for the user and a practical project for the developer. A tool must meet the majority of the criteria to be chosen for integration. The tool criteria are listed below in order of importance, from highest to lowest

1. Executes on a Unix platform
2. Enhances *AFIT*tool by fixing one or more shortcomings
3. Provides ease of integration
4. Has technical support available
5. Has reasonable and acceptable cost
6. Assists the user in understanding information stored in *AFIT*tool

Ideally, the tools to be integrated would be based on Lisp or Refine so they could be easily integrated into *AFIT*tool. Since *AFIT*tool is run on a Unix platform, it follows that it is preferable for any integrated tools to run on the same platform. Additionally, if the tool is a commercial product, a high level of technical support is desired. If it is not commercial, it is important that the developers provide technical support. If the tool is not well supported and/or documented by technical expertise, it is not a likely candidate for integration. The tools need to be available for AFIT to either purchase at a reasonable cost or acquire free of charge. Finally, the tools chosen for integration need to assist the user in understanding, and possibly verifying, the information currently stored in *AFIT*tool. The following subsections discuss different types of tools that were considered for integration.

2.3.3 Architecture tools. There are many tools available to define and manipulate the architecture of a software system. The following tools were developed at universities under the Evolutionary Development of Complex Software (EDCS) project. They have become strong tools, and support some level of integration with each other through the Interface Definition Language Acme.

- Aesop: Aesop provides a toolset for constructing open, architectural environments that support architectural styles. It interfaces with other tools through a Remote Procedure Call (RPC) interface, allowing other tools to analyze and manipulate architectural descriptions. Aesop would enhance *AFIT*tool by offering another capability — specifying the architecture of the system being developed. Aesop is implemented on a Unix platform and is available for release as a demonstration system from Carnegie Mellon University [11].
- Acme: Acme has been developed to provide a common ground for software architectures. It can be used for developing a system architecture as well as interfacing multiple architecture designs. Currently, it supports translation between UniCon and Aesop, as well as from Wright to Rapide. Acme would enhance *AFIT*tool by allowing the user to specify the architecture of the system being developed and translate between architectures, if desired. Acme is implemented on a Unix platform and can be obtained from Carnegie Mellon University [12].
- Rapide: The Rapide project is an effort to develop new technology for specifying the architectures of component-based large-scale, distributed multi-language systems. The toolkit available to work with the Rapide language allows gradual refinement of the architecture, thereby supporting incremental development, testing and maintenance. Rapide is another tool that could be integrated with *AFIT*tool to allow the user the capability of specifying an architecture. The Rapide toolkit executes on the Unix platform. This research is being performed at Stanford University, hence all information and toolkits can be obtained without cost [17].
- Wright: Wright provides a formal basis for architectural description. It is an architecture description language that allows the user to describe an architecture with

precision. It also enables the user to analyze both the architecture of individual software systems and of families of systems. Additionally, Wright defines consistency checks that the user can perform to increase their confidence in the design of a system. Wright is another alternative to allow the introduction of architecture descriptions into *AFIT*tool. The Wright project is being pursued by researchers at Carnegie Mellon University. They are currently developing a toolkit on the Unix platform to work with the Wright language and it would be available at no cost to AFIT [18].

2.3.4 Software Development Environment tools. One approach to tool integration involves starting with a development environment and expanding it to create a custom software development environment. The tools discussed here are Computer-Aided Software Engineering (CASE) tools that can further enhance *AFIT*tool by offering a less formal view of the system. Additionally, these tools support the whole software life cycle, meaning they can be used at all stages of the development process.

- Rational Rose: Rational Rose is a development environment that allows software systems to be developed. It has many features, including language development in Ada95, Java and C++, Corba/IDL generation, database schema generation and an extensibility interface. Rose 98 supports the Unified Modeling Language (UML), Booch, and Rumbaugh notations making it more flexible. It has been implemented for both the PC and Unix platform. AFIT currently owns an educational license for Rose, making a purchase unnecessary. Integration of Rose with *AFIT*tool could be accomplished through Rose Scripts or Rose Automation. This integration would offer a visual modeling tool that is currently unavailable to the user and would help the user and the customer understand what is being modeled [21].
- Knowledge Based Software Assistant Advanced Development Model (KBSA ADM): ADM, developed for Rome Labs by Andersen Consulting, is a tool that was designed to encompass the entire software life cycle. It provides an integrated environment that could possibly be used as a common ground to integrate *AFIT*tool with other tools. ADM incorporated Object Store for persistence as well as other tools/languages for

requirements acquisition and project management. It uses both the Unix and Windows NT platforms to achieve its goals. AFIT acquired ADM through the Rome Site of Air Force Research Laboratory, making it unnecessary to purchase this software [8].

2.3.5 Drawing/Diagramming tools. This section describes several drawing tools available in the Unix environment. One way, out of the many possibilities, to use a drawing tool is to graphically depict the state transition diagram. The tools could also be used to allow the system to display the object model or the event flow diagram for the current domain. Beyond simply drawing or displaying, it may also be possible to animate the drawings to show the progression of the state transition table.

- xfig: xfig is the drawing tool that is standard with Unix. Users can draw pictures and save them as .fig files or export them to .ps files. It is a primitive tool but may be a possibility for integration since it is currently available at AFIT and runs on the Unix platform.
- Graph Layout Toolkit: The Graph Layout Toolkit was developed by Tom Sawyer Software to develop graphs in a GUI environment. It appears that there is also a Java API that is used to interface with the software. By using the Java API, it is available for Unix and PC platforms. This commercially available software can be purchased at an educational price [22].
- daVinci: daVinci is a tool developed by the University of Bremen on the Unix platform for graphical layout of nodes. daVinci has been integrated into the Artificial Intelligence system developed at AFIT called PESKI using an interactive remote procedure call interface. daVinci also supports command line options, allowing the user to specify a graph to draw using term notation. This tool is available at no charge and is currently on an AFIT system [10].
- Island Draw: Island Draw is a drawing tool that allows the user to create high-quality diagrams. It provides an import and export facility, allowing many formats to interact with the program. For example, it is possible to import a postscript file into Island Draw and export an Island Draw file. Island Draw is available at AFIT [19].

2.3.6 Theorem Provers. The following tools support proving the correctness, with respect to supplied pre- and post-conditions, of specifications and code. The tools require their input to be in a specified format or language, specific to the tool. These tools could offer the ability to prove the correctness of a specification, before the transformations are completed to generate code. Additionally, the tools would assist in proving the correctness of the resulting code.

- Epilog Inference Package: Epilog is a library of Common Lisp subroutines to be used in programs that manipulate Standard Information Format (SIF) files, a variant of first order predicate calculus. It has built in routines to convert expressions, do various pattern matching, and create and maintain SIF knowledge bases. To be integrated with *AFIT*tool, a converter would have to be written to put the domain model (or other file to be examined) into SIF. Since Epilog uses Common Lisp, and Refine is built on Common Lisp, the rest of the integration should be straightforward. Epilog was built for Macintosh and Unix and is available for download from Stanford University [16].
- Z/Eves: Z/Eves is a theorem prover that supports the analysis of Z specifications by performing syntax and type checking, schema expansion, precondition calculation, domain checking and general theorem proving. Since *AFIT*tool currently uses Z specifications to represent the requirements, this tool could enhance *AFIT*tool by allowing these specifications to be proven. Additionally, the files needed to integrate with *AFIT*tool are already input to the system, so integrating Z/Eves with *AFIT*tool would be straightforward. Version 1.5 of Z/Eves is available at no cost from ORA Canada. It runs on SunOS, Object Store/2, Linux, Windows 3.1, Windows 95 and Sun Solaris [24].
- ProofPower (supports HOL): ProofPower is a commercially available tool that supports the proof of HOL and Z specifications. The Z specifications are entered as LaTeX files, just as they are entered into *AFIT*tool. This would make it possible to use the same files as the interface between ProofPower and *AFIT*tool. It is

available for purchase from ICL under an educational license and runs on the Unix platform [20].

- AMN-PROOF (supports HOL and PVS): AMN-PROOF is a theorem prover that supports proof of Abstract Machine Notation (AMN) specifications and refinements with the HOL and PVS theorem provers. A system is specified as a number of abstract machines that are formally refined to an executable representation in terms of more abstract machines. In the present version, the HOL prover is not supported [15]. The tool is written in C++ and is available for Linux and Sun Sparc machines by free download. This tool seems to be very rudimentary and may not be a good choice for integration to *AFIT*tool.

2.3.7 Data Storage. The following two tools are possibilities for adding persistent storage to *AFIT*tool. This storage would be used to store the domain model and should offer a querying capability. Storing and loading domain models would increase ease of use and reuse in the system, saving users effort.

- Object Design's Object Store: Object Store is an Object-Oriented Database Management System developed by Object Design. This could serve as the repository for the domain models *AFIT*tool produces. It supports a CORBA-compliant interface, allowing integration with external tools. In order to integrate *AFIT*tool with Object Store, the domain AST would need to be output in a CORBA-compliant format. Although not trivial, this could be accomplished. AFIT currently owns Object Store, so a purchase would not be required. It runs on both Unix and PC platforms.
- Refine's Persistent Object Storage: Refine has a built-in capability to store the AST in a file called a Persistent Object Base. This must be programmed in Refine, specifying each node of the AST. Currently, *AFIT*tool has this functionality in a limited manner, but it is easily extendible. AFIT currently owns Refine and *AFIT*tool is based in Refine. It runs on the Unix platform.

2.4 A Sampling of Integration Methods

There are a multitude of methods available to integrate a tool set. One method is to use a scripting language to handle the control flow of tool execution, allowing user input when necessary. The script would also send any necessary commands and/or data to the applications. Scripting languages are commonly used as the "glue" for an integrated environment in many application areas. Developed components are integrated into applications using a scripting language [27]. The most common languages are tcl and Perl, but there are many others available for use, including Visual Basic and JavaScript [27]. They are often used when control flow integration is desired. The most common use of scripts involves a script that controls what is to be executed, what data is passed and what the user sees on the screen. The use of scripts can often speed up development time by a factor of five to 10 [27]. Tcl can be used in two ways: as a method for building application interfaces and as a uniform framework for communication between tools [26]. Tcl has some of the characteristics of Lisp, but it was designed to be embedded in an application program, rather than to develop stand-alone programs [26].

A related concept is the use of intelligent agents, with a script, to perform the required tasks [13]. Agents are used as a communication and control mechanism between software components to create an integrated environment. Agents have been compared to objects, and have some similarities, such as a message-based interface independent of the internal data structures and algorithms. In programming efforts involving agents, an agent communication language is needed, as well as constructs to allow the agents to communicate within the framework of the system and the language. One agent language in widespread use is called Knowledge Query and Manipulation Language (KQML). It uses the idea of a message, which the agent can send and receive [13]. Agents, then, can control both the flow of execution and the flow of data in a system. Because of these properties, they are very useful in system integration. The agents must be controlled by a higher level process, or a server of some sort, in order to have the proper flow in the system.

Another possibility for integration is to use an existing CASE tool, such as ADM or Rational Rose, as the basis and integrate add-ins to that tool. This would extend an already integrated environment to provide additional functionality to the users. Since the

majority of the environment is in production, the integration of more tools would simply enhance the tool, rather than redefine it.

With the growing dominance of Object-Oriented (OO) software development, two standards have emerged to work with the data: Common Object Request Broker Architecture (CORBA) and Distributed Component Object Model (DCOM). The fundamental idea is that a middleware application will be used to allow other applications to interact, without having the same data types or formats. All data is stored as objects, in accordance with the OO paradigm, and methods are used to access the data. In the CORBA realm, the middleware is called an Object Request Broker (ORB) and the data is stored and retrieved using ORB methods. Each client must register with the ORB in order to use it. The client would then send requests to the ORB to store, retrieve or manipulate the data. Most ORBs have a query facility built in to find the desired data. The Object Management Group, composed of business leaders in the OO community, developed CORBA and it is available on both the PC and Unix platforms [2] [4]. DCOM has similar constructs for common objects. DCOM was developed by Microsoft Corporation and is available only on the PC platform [3] [4].

The emerging method of tool integration seems to be to use some sort of "middleware" that ties tools together in the background, without the explicit knowledge of the user. This method can also use several of the previously mentioned methods, including RPCs, message passing and a CORBA interface. The "middleware" would provide a common ground, as CORBA does for objects, from which all applications are run. This method could also make use of intelligent agents that would perform the requested tasks and return with the data or a message to the starting point. Web browsers are sometimes used to integrate different applications by using a CGI application in the HTML code of the web page. This has been very successful in the database arena, allowing companies to provide on-line purchasing through a web page that communicates with a database [1]. Additionally, the user is unaware of the true interface to the system, since they see only the web page.

High-level programming languages such as Visual Basic and C++ are often used as driver programs to control the flow of execution and data throughout a system. Some applications include a menu system that allows a user to choose each task that is accom-

plished, while other applications allow the user to start the system and several tasks are accomplished in the background without user intervention. It seems that for an effort of any size, a combination of methods will be used. Rather than accomplishing the whole system integration through message passing or a CORBA object repository, a combination of communication and control flow methods will be used. The main advantage of combining methods is flexibility. It allows tools to be located on different platforms, written in different languages, and have different user interfaces.

2.5 Tool Integration Models

Over the years, the literature has provided documentation on many different ways to approach tool integration. The most widely adhered to model was developed by Anthony Wasserman and includes five classes of integration: platform, presentation, data, control, and process [34]. One team of researchers discarded platform integration, arguing that the primary focus is on the relationship between tools [34]. Other researchers have discarded platform and process integration, leaving presentation, data, and control integration [35].

2.5.1 Thomas and Nejmeh's Approach to Wasserman's model. Thomas and Nejmeh discarded platform integration from Wasserman's model, arguing that the relationship between the tools is the most important issue, while the platform provides the basic building blocks for integration [34]. The following sections describe the four classes of integration Thomas and Nejmeh discuss, along with properties they identified for each class.

2.5.1.1 Presentation Integration. Presentation integration deals with the relationship between user interfaces of tools. A highly integrated environment, from the standpoint of presentation integration, is one which does not force the user to understand multiple interfaces. Two properties have been identified in presentation integration: appearance and behavior, and interaction-paradigm integration [34].

1. Appearance and behavior: This property addresses the ease of use of the integrated toolset. If a user understands one tool, how does that knowledge help them in dealing

with other tools in the environment? Two tools that are well integrated with respect to appearance and behavior allow the user's experience with and expectations of one tool to apply to other tools. Appearance and behavior integration captures both lexical and syntactical similarities and differences in tools. The lexical elements of a tool include things such as how the mouse clicks, how the menu bar looks, where windows are placed, and if there are multiple windows or just one. Syntactical elements of a tool include the order of commands and parameters, presentation of choices in a dialog box, and the format of input and output files. Although windowing tools are influenced by the guidelines of Motif and OpenLook, there is enough flexibility to allow ambiguity in an integration effort.

2. Interaction-paradigm: The interaction of two tools can be very similar or very different, and the degree of difference impacts the user by causing him or her to have to learn a great amount about the interface of a tool, if they are very different, or not learn much at all, if they are similar. The underlying metaphors and mental models of the tools are the two primary factors in interaction-paradigm integration. The two tools are well integrated with respect to interaction-paradigm if they use the same metaphors and mental models. The use of only one metaphor for the entire system may be unachievable, but a balance is key. Two tools that use similar file navigation methods are said to use similar metaphors. For instance, most MS Windows systems use a file hierarchy approach to file navigation. Another approach is that of a hypertext structure, where files are displayed and there is no emphasis on which files are contained in which higher structures. These two common methods of file navigation use very different metaphors. Integrating two tools using these two different methods will result in an environment that is not well integrated with respect to interaction-paradigm integration, without changes.

2.5.1.2 Data Integration. Thomas and Nejmeh identified five properties of data integration, defined over the data management and representation aspects of two tools: interoperability, nonredundancy, data consistency, data exchange, and synchronization [34].

1. Interoperability: This property addresses the issues of two tools needing the same data, and needing to view it in the same way. In some cases, the data may be semantically correct overall for both tools, but the tools may attach different semantics to the same data. This aspect of data integration, when addressed, answers questions regarding what has to be done to make the data available and correct for both tools. The best scenario for integration, based on data interoperability, is two tools using the same model and format. Two tools that use the same type of data, but expect it in completely different formats, are not as well integrated and require a data conversion program. Interoperability applies to persistent data only.

2. Nonredundancy: Nonredundancy describes the desire that two tools have little or no overlapping data. This aspect of data integration examines the amount of overlapping data that each tool stores and manipulates independently. If two tools have data that is exactly the same, or can be derived from other data, it is difficult to ensure consistency of the data in the integrated system. Therefore, it is desirable to minimize redundant data. It may be practical, however, to have replicated data in a database to improve performance.

3. Data consistency: Maintaining consistency of redundant or derived data may involve special semantic constraints on the data involving the interaction of two tools. For instance, two tools may have data independent of each other, but when integrated, the data of one tool relates to the other in such a way that the admissible values are restricted. When integrating tools in which this is applicable, it is important for the tools to cooperate to maintain the semantic constraints on the data. This requires each tool to "report" its data manipulations and their effects to other tools.

4. Data exchange: Some tools may need to exchange data, whether it be initial values at the start of execution or updated values during execution. When two tools such as these are integrated, the integration effort must address what needs to be done to data generated by one tool in order for another tool to manipulate it. Data exchange involves the tools agreeing on semantics and data format. If little or nothing needs to be done to the data in order to exchange it between tools, the tools are well integrated with respect to data exchange. Although this is similar to interoperability

of data, it also applies to nonpersistent data and may use different mechanisms to share the data.

5. Synchronization: Synchronization is mainly concerned with the consistency of nonpersistent data shared among tools. Maintaining consistency involves each tool communicating any changes made to the data to all other tools. Since most tools will use both persistent and nonpersistent data, synchronization is often an issue. Although it is very similar to data consistency, synchronization applies to nonpersistent data, while data consistency applies to persistent data.

2.5.1.3 Control Integration. One goal of a well-integrated toolset is to share functionality between tools in such a way that the user gains access to the full functionality without knowing which tool owns the functionality. Sharing functionality requires tools to pass control from one to another, knowing only what functionality is needed, not which tool is needed. Additionally, tools must communicate the operations to be performed when passing control. Control integration complements data integration in that to pass control from one tool to another, data or a data reference is often needed as well. Thomas and Nejmeh identified two properties of control integration: provision and use [34].

1. Provision: Provision describes the extent to which each tool is needed by the integrated environment as a whole. If a tool is added to the environment but not used, it is said to be poorly integrated with respect to provision integration. Alternatively, a tool is well integrated if it offers services other tools in the environment will use.
2. Use: The property of use complements that of provision in that it measures the extent to which a tool uses services offered by other tools in the environment. In order to achieve high use integration, the tools must be modular. Additionally, each tool in the environment must use the services provided by other tools rather than supplying the services within the tool.

2.5.1.4 Process Integration. Process integration is the fourth class of integration discussed in Thomas and Nejmeh's article. It deals with ensuring tools interact well to support a defined software process. Tools that support a software process make as-

sumptions about that process. Tools are said to be well integrated with respect to process integration if these assumptions are consistent. There are three properties associated with process integration: process step, process event and process constraint. An integration effort only needs to address process integration if the tools being integrated are relevant to the same process step; for example, they both influence requirements analysis.

1. Process step: A process step is equivalent to a phase in the software life cycle. This property addresses how well the tools combine to support a step in the process. In an integration effort, one process step may be broken into smaller steps, each related to a tool. These tools usually work in sequence, the execution of one satisfying the precondition of the next in such a way that it may achieve its goals. Tools are well integrated with respect to process step integration if the integrated tools complete the process step and allow other tools to do their work. Conversely, tools are said to be poorly integrated if one tool makes it harder for other tools to achieve their goals, or if one tool does not satisfy the precondition directly, causing more work to be accomplished by other tools.

2. Process event: A process step is composed of process events that, when executed sequentially, achieve the goals of that step. From the standpoint of tool integration, the integration is measured on how well the tools agree on the events that need to be accomplished in each step. Thomas and Nejmeh identified two aspects of process event agreement. First, the preconditions of one tool should reflect events generated by other tools. Second, one tool should generate events that help satisfy the preconditions of other tools. Tools are well integrated with respect to process event integration if they generate and handle events consistently.

3. Process constraint: A process constraint is a condition that restricts some aspect of the process. In a tool integration effort, process constraints are examined to determine how well they cooperate to uphold the constraints. Tools are examined both on whether or not their functions are constrained by another tool's functions and if their functions constrain another tool's functions. If two tools agree on the

range of constraints they recognize and respect, they are well integrated with respect to process constraints.

2.5.2 Wallnau and Feiler's Approach to Wasserman's Model. Wallnau and Feiler refine Wasserman's model differently than Thomas and Nejmeh. They argue that "framework and process integration are orthogonal to control, data, and presentation integration (and to each other)" [35]. In their opinion, process integration defines what tools get integrated, while framework integration defines how tools get integrated. This is effectively the same relationship that exists between requirements and design in a software development effort. Additionally, Wallnau and Feiler believe the original model that deals only with control, data and presentation integration is sufficient to characterize tool integration. They do not believe, however, that control, data and presentation relationships define how to integrate tools, but rather they describe the relationships between integrated tools. Therefore, although the framework provides mechanisms to integrate tools, the tools themselves provide the details of integration. They view integration as being composed of three classes of entities: framework, process, and tools [35]. Since this research effort is focused on tool integration, that is the only class described below. Intertool integration is viewed as having three distinct types of integration, namely control, data, and presentation integration.

2.5.2.1 Control Integration. Control integration embodies the concept of one tool executing functions of another tool, or supporting remote execution of functions of a tool. Control integration can be used to move control to where the data resides, rather than taking the data integration approach of moving data to where control resides. In the past, integration efforts have used data-driven and event-driven triggers, much like database triggers. Data-driven triggers cause actions to occur due to a change in data, such as a change in a database. Event-driven triggers cause actions to occur due to a change in the environment. The significance of recent development in control integration is the execution of a tool's lower-level functions.

2.5.2.2 Data Integration. Data integration has been the most common class of integration in research efforts for many years. The central point of many research efforts is making the data models and format agree between the tools. One way to achieve this goal is to have data in a central repository and manipulate the tools to access this data. Three other possible methods for data integration are format mechanisms, storage mechanisms and carrier mechanisms. Format mechanisms use an agreed-upon format for communication between tools, and include solutions that use a non-proprietary external format for data, such as PostScript. Solutions that use storage mechanisms for data integration often involve using common databases, clipboards, and external files for data sharing. Carrier mechanisms, such as pipes and sockets or remote procedure calls, are also used for data integration. Agents carrying data could also be carrier mechanisms.

Research efforts in the area of data integration have determined that some data repository services are often needed by the environment, to accomplish configuration management and project management, and have nothing to do with tool integration. Due to this requirement, research has continued toward developing a central repository to handle data integration in CASE environments. This research has promoted the development of two concepts related to repository and data integration. The first is that the repository data model and the data management services should be separate. This approach allows the services and repository to access objects through the data model, while other environment tools can access the data management services directly. This concept supports a layered model for data integration. The second concept supports separating relationship management services and data management services. The key behind this concept is separating the relationship services from the underlying data model. This concept is aimed at providing traceability and configuration management services to tools that manage their own data.

2.5.2.3 Presentation Integration. The goal of presentation integration is to provide a common look and feel for the integrated system. Over the years, this has become less platform dependent, meaning X Windows applications and Macintosh applications may have the same kind of user interface. In order to achieve presentation

integration, tools must agree on a standard interface, an ideal that is not often practiced. Another approach is to use a user interface management system (UIMS), causing the system to be less dependent on low-level mechanisms and more dependent on window system-independent mechanisms. UIMS offer a framework for the integration, presenting a common look and feel to the integration effort without changing the underlying tools. Integrated Project Support Environment (IPSE) frameworks are in the same class as UIMS, and will not be widely used, for much the same reasons. Although both seem like a very promising concepts to presentation integration, they have not been widely adopted due to their immaturity and lack of availability in customer environments.

2.6 Summary

This chapter presented many tools that were candidates for integration with *AFIT*tool, as well as the set of criteria that was used to select tools. Additionally, some of the methods of integrating tools as well as the models upon which integration can be based were discussed. The next chapter presents a methodology for integrating software tools which was influenced by Thomas and Nejmeh's characteristics of data and control integration.

III. Tool Integration Methodology

The integration of two or more tools is intended to form a complete system, with the goals of the user in mind. In most cases, tools are integrated in order to form a system that supports a software development process or another business practice, such as managing inventory. Tool integration may become necessary after the merger of two companies, each one using different software to manage inventory. After the companies merge, the new company will need a single method of managing inventory and will not want to lose inventory information or functionality provided by the two tools. Integrating the two tools allows the new company to take advantage of all of the functionality of the two tools with the interface of one tool. The data that was previously used by two separate tools can also be transformed to a central repository which can be used by the integrated system.

Since the goal of tool integration is to form a fully operational, totally integrated system, it is important to have a framework of rules guiding the integration process. For this reason, the primary goal of this research has been to develop a generic methodology that covers the majority of tool integrations involving existing tools. This methodology is based on the concept of a ***design space***, composed of functional and structural dimensions. The ***functional dimensions*** of the design space identify the requirements for the tools that most affect the solution for the integration effort. The ***structural dimensions*** of the design space determine the overall framework of the integrated system. These dimensions detail the characteristics of the pair of tools being integrated (functional dimensions) and the characteristics of the resulting system (structural dimensions). This concept of a design space was discussed by Lane in his work on software architectures [25]. A design space is one method of classifying tools by examining each dimension. Each dimension enumerates all of the possibilities for that aspect of a tool [25]. The mapping from the functional dimensions to the structural dimensions is achieved through the use of ***design rules***, guidelines for choosing between structural dimensions, given a set of functional dimensions.

Although there are many methods used for tool integration, the methodology in this chapter offers a way of choosing between methods and achieving the goal of an integrated system. In developing this methodology, Lane's concept of a design space was extended. The concept presented by Lane included one set of functional dimensions and one set of

structural dimensions. For this effort, however, it seemed appropriate to have two sets of functional dimensions: one for a single tool and one for the tool pair. The methodology is based on this extension, rather than the strict model presented by Lane [25]. The chapter begins with an overview of the methodology, followed by a description of the functional dimensions and structural dimensions. Finally, the chapter concludes with the design rules for this design space.

3.1 Methodology Overview

In order to integrate two software tools, it is necessary to characterize the tools both individually and as a pair. The aspects of a single tool that apply to the integration include the methods used for data input and output and whether or not the tool can be extended, and if so, how it is extended. These aspects are classified as functional dimensions of a single tool, explained in the next section. The first step in the integration effort is to determine the values of each of these aspects. The next step is to examine the interface of the tool pair. The extendability of the two tools collectively is examined, as well as whether or not the data they share is compatible. These aspects are captured in the functional dimensions of a tool pair.

After determining the values of each of the functional dimensions, the design rules for the design space can be applied, yielding values for each of the structural dimensions. The structural dimensions include determining the communication path that will be used in the integrated system, as well as the method of transforming the data and controlling the system. These steps are summarized in Table 1. The following sections describe each of the dimensions in detail, including the alternatives (values) for each dimension.

3.2 Functional Dimensions

Tool integration encompasses the tools in their entirety, but only certain aspects of the tools are actually considered in the integration. The functional dimensions for the tool integration design space take into account the characteristics relevant to the integration of the tools. After examining the data integration characteristics proposed by Thomas and Nejmeh [34], two sets of functional dimensions were defined. Section 3.2.1 specifies the

Table 1 Methodology for Tool Integration

Step 1:	Determine for each tool: Input Mechanism Output Mechanism Extendability
Step 2:	Determine for each tool pair: Extendability Class Data Compatibility
Step 3:	Apply design rules to determine structure of system Provide output of first tool and input of second tool to determine communication path. Apply design rules based on extendability class to determine control integration implementation and data transformation.

characteristics of a single tool, the first set of functional dimensions, while Section 3.2.2 characterizes a tool pair, the second set of functional dimensions.

3.2.1 Functional Dimensions for a Single of Tool. The important aspects of a single tool are the method it uses for input, the method it uses for output, and the how the tool can be extended, if possible. To characterize a tool, one alternative from each dimension is chosen. The following paragraphs describe the three functional dimensions of a tool: input characteristics, output characteristics, and tool extendability, illustrated in Table 2.

3.2.1.1 Input Characteristics. The first functional dimension for a single tool, input characteristics, characterizes the kind of input a tool uses. The input of a tool is characterized by the method it uses to obtain the data it needs. The input data is used to support the tool's functionality. The input data sources of a tool can be persistent or non-persistent, and any given tool may use more than one of the alternatives offered below. For tool integration, however, the tool is characterized by the input mechanism that is relevant to the integration effort. Tools that accept input through the use of a graphical user interface (GUI) may be integrated using this methodology, but the methodology does not specifically address concerns that may arise when integrating a tool with a GUI. For instance, the desire to create a seamlessly integrated system, with the *appearance* of one tool is not addressed with respect to a tool with a GUI since that deals with presentation

Table 2 Functional Dimensions for a Single of Tool

Input Characteristics
Standard Input
File
Command Line Parameters
Message Passing
Output Characteristics
Standard Output
File
Message Passing
Built-In Output
Tool Extendability
Source Code available
Tool Provides Extension
Both
Neither

integration. If the tool has a GUI but also accepts input from another source, that is how it should be characterized. There are four alternatives for the input dimension of a tool, as follows.

- Standard Input (stdin): Stdin is the default input mechanism for many applications and supports operating system redirection. In command line applications, stdin refers to what is entered from the keyboard after the tool begins to execute.
- File: The data needed by the tool is held in a file. The tool knows the format of the file and uses it for the data requirements. The path and name of the file may be stored internal to the tool or may be supplied by the user at run-time. For the purposes of this research, any persistent data source is termed a "file."
- Command line parameters: Command line parameters are parameters supplied when the tool is executed. A tool could accept small amounts of data on the command line, as in `doquery -T<text>`.
- Message Passing: If input from a tool is accomplished via message passing, it expects data and possibly control messages in a certain format. These messages give the tool the information it needs to perform the functions requested by the user. A tool

that uses messages as input makes it possible to monitor the flow of communication, capturing key messages for use in controlling the system.

3.2.1.2 Output Characteristics. Output characteristics of a tool, the second functional dimension, describe the methods a tool uses to record the results of the functions it performs. Sometimes this "recording" is persistent, as in the case of a file, and other times it is not, as in the case of standard output. The possibilities for output from a tool are very similar to input, but there are some differences. For instance, the printer and standard error are legitimate output destinations, but are not often considered input sources. Similar to the input characteristics, a GUI is a valid output mechanism for a tool, but is not considered in this methodology. There are four alternatives to this dimension.

- Standard Output (stdout): Stdout is the default output mechanism for many applications, much like stdin is the default input. Stdout refers to data or messages printed to the screen. Additionally, stdout supports operating system redirection.
- File: The tool writes its output to a known file location and format. The location of the file may be specified internal to the tool or it may be supplied by the user at run-time.
- Message Passing: Using message passing as an output source involves the tool sending formatted messages containing data to an external destination.
- Built-In Output: There are several output streams available to some tools through the operating system or the chosen programming language, including the printer, standard error (stderr) and a log file. The printer is usually reserved as an output destination for formatted data. Stderr is the default output mechanism for errors. It is used by both GUI and command line applications. The log file is an output destination for messages the tool writes for the user's benefit. Additionally, some tools have an internal format that is used to store data, and is not actually output, such as the ASTs produced in Refine. These built-in output mechanisms may or may not support operating system redirection.

3.2.1.3 Tool Extendability. The third functional dimension, the extendability of a tool, is an important functional dimension in the design space. The ability to extend the tool means the integrator is able to extend the tool beyond its current capabilities. If the source code is available, routines may be written for data integration, control integration, or both, and compiled into the tool so that they become part of the tool. Data integration routines may be written to pre-process or post-process the data.

If the source code is not available but the tool is extendable by some other method, the same kinds of routines may be written, as described previously, but they will not be part of the compiled version of the tool. Tools that offer this sort of extendability often accomplish it through a combination of allowing menus to be extended (or added) and some sort of programming language. This method gives the integration expert the ability to define a menu option for the desired functionality and write a script or program to achieve the functionality. The last possibility is that the tool cannot be extended. There are four alternatives to this dimension: the source code is available, the tool provides for extension, both, or neither.

Table 3 Functional Dimensions for a Pair of Tools

Extendability Class
Neither Extendable
First Extendable
Second Extendable
Both Extendable
Data Compatibilty
Compatible
Not Compatible

3.2.2 Functional Dimensions for a Pair of Tools. The functional dimensions for a pair of tools are necessary in order to characterize the interface between the tools. The tool interface is central to the integration effort and must be fully understood in order to proceed with the integration. When examining the tool pair to be integrated, the extendability of the *pair* is important, as is the data compatibility between the tools. From the values of these dimensions, the design rules can be applied and the structure of the integrated system can be determined. The tool pair is considered as having a

"first" and "second" tool, indicating the direction of the flow of data. In the case where tools communicate in two directions, i.e., the output of tool A is used by tool B and vice versa, this methodology should be applied twice, once for each direction of data flow. The following sections describe each dimension, the extendability class and data compatibility of a tool pair, illustrated in Table 3.

3.2.2.1 Extendability Class. In examining a tool pair, the first functional dimension involves determining the extendability of the tool pair. This dimension considers all possible combinations of the *Tool Extendability* dimension for one tool, yielding 16 possibilities for the extendability of a tool pair. These possibilities can be grouped based on similarities in how the integration is accomplished. For instance, there is not a difference between the integration of a tool pair with the Extendability values *Source Code* and *Tool Provides* and the integration of a tool pair with values *Source Code* and *Source Code*. Both of the tool pairs have two extendable tools. Therefore, since the tool pairs fit into the same equivalence class with respect to integration, the design space was reduced to reflect the similarities of the tool pairs. Following this guideline, the four alternatives to this dimension are produced: Neither Extendable, First Extendable, Second Extendable, and Both Extendable. Table 4 illustrates how the possibilities for the individual tools combine to determine the Extendability Class for the tool pair.

Table 4 Extendability Class

	Tool 2			
Tool 1	Neither	Source Code	Tool Provides	Both
Neither	Neither	Second	Second	Second
Source Code	First	Both	Both	Both
Tool Provides	First	Both	Both	Both
Both	First	Both	Both	Both

3.2.2.2 Data Compatibility. The second functional dimension for a tool pair, Data Compatibility, describes whether or not the data that the tools share is compatible. Data can be compatible syntactically and/or informationally. It is assumed that the data is compatible informationally, meaning the data required by one tool is represented in some form by the other tool. Therefore, this dimension needs only to characterize whether

Table 5 Structural Dimensions

Communication Path
Shared Data
Data passed via stdin/stdout
Data passed via message passing
Data passed via middleware
Control Integration Implementation
Client-Server
Centralized
Distributed
Data Transformation
Transformation by output tool
Transformation by input tool
Transformation by both tools
Transformation by external source
No Transformation needed

or not the syntax of the data is compatible. If the data for both tools is in the same format, the data is compatible sytactically. The compatibility of the data influences the structure of the integrated system, specifically when data integration is performed. The alternatives to this dimension are compatible and not compatible.

3.3 Structural Dimensions

The *structural dimensions* of the design space represent the outcome of the decisions made as a result of analyzing the functional dimensions, determining the integration methods of the overall system. The structural dimensions are: Communication Path, Control Integration Implementation, and Data Transformation, illustrated in Table 5.

3.3.1 Communication Path. The first structural dimension, the communication path, describes the manner in which the data is exchanged by the tool pair being examined. This dimension of the design space has four alternatives: shared data, data passed via stdin/stdout, data passed via message passing and data passed by middleware.

- Shared Data: In an integrated system, several tools often use the same data, with each tool reading from and/or writing to the data. There are several possible storage mechanisms for shared data, including a file, common objects, or a database. The

key characteristic of shared data that distinguishes it from another type of data is that it is persistent, allowing one tool or several tools to access the data. Since multiple tools are working on the same data, problems with synchronous data access could be encountered including resource locking, stale data, and timing issues. These problems have to be addressed during the integration effort.

- Passed Data: Data that is needed by the second tool may be sent from the first tool so that they may both use the data. The data passed is generally non-persistent, relevant only during the execution of the tools using the data. Three of the alternatives of this dimension are special cases of passed data. The communication path for the data can be through stdin/stdout, message passing, or via middleware. The characteristics of stdin/stdout and message passing described in Sections 3.2.1.1 and 3.2.1.2 are valid in the case of passing data as well as input and output mechanisms for the tool.

3.3.2 Control Integration Implementation. Control integration, the next structural dimension, provides a seamlessly integrated system by managing the control flow for the entire system [9]. When several tools are integrated into one environment, one of the goals of the integration effort is to develop an environment with a logical control flow, based on the user's criteria, such as supporting a particular software process or methodology. Control integration can provide the illusion of a single system consisting of multiple components rather than a system that is an aggregate of several tools. Control integration implementation has three alternatives: client-server control, centralized control, and distributed control.

- Client-Server Control: Client-server control involves one tool invoking another tool. Based on the characteristics of the system, client-server control may involve executing another tool or it may involve simply executing specific functions of another tool.
- Centralized Control: Centralized control is usually achieved through the use of one driver program. This program executes the tools at the appropriate times. It may also need to run any necessary data conversions and pass data and messages between programs. (See Section 3.3.3.) The user sees one system, through the driver program.

- Distributed Control: Distributed control in an integrated system means control is spread throughout the system, requiring several components of the system to handle control. One approach to distributed control is to develop wrappers or individual controllers for each tool to handle some aspect of control for the system. Each wrapper would handle all communication from the other wrappers and invoke the tool when necessary. Specifically, agents can be used as wrappers to handle tasks such as communicating with other tools, running another tool, or running a data conversion routine before invoking another tool. Agents are usually passive monitors that only become active when a trigger event occurs, such as one tool completing its write to a central data source [13]. The agent may then be programmed to notify the other tool(s) in the system that they can use the data. Agents are often used as a glue for integrated systems. The agents will actually act more like messengers in that they can carry data or control information and wait for a reply before returning.

3.3.3 Data Transformation. The final structural dimension, data transformation, addresses the amount of similarity of the two tools' data before integration. Two tools that use the same data can be evaluated on the degree of transformation the data must undergo before it can be used. This dimension has five alternatives: no data transformation needed, data transformation performed by the output tool, data transformation performed by the input tool, data transformation performed by both tools or data transformation performed by an external source.

- No transformation needed: Data that does not need to be transformed is in the proper format for both tools without any translation.
- Transformation performed by the output tool: As part of the integration effort, the choice may be made to use a tool's input data format as the common format for integration. In this case, any tool which outputs data must post-process this data, converting it to the chosen format.
- Transformation performed by the input tool: A tool using data from a common data source may need to pre-process it to the format it expects before it is used.

- Transformation performed by both tools: If two tools use the same data, it may be the case that the data is stored in a standard format, causing both tools to transform the data to and from the standard format.
- Transformation performed by an external source: Tools may rely on an external program to transform data into a format they can use. Transformations performed by middleware could be accomplished with scripts or a program in a high-level language.

The functional and structural dimensions of the tool integration design space have now been described in detail. These dimensions are used to describe the inputs and output of the methodology: the tools which are to be integrated and the resulting integrated system. The following section discusses the design rules of the design space, which provide a mapping from the structural dimensions to the functional dimensions of the design space.

3.4 Design Rules

Design rules are meant to be guidelines used by the integrator to decide which method of integration to choose. For the tool integration design space, there are a potential multitude of design rules. The rules in the following sections were developed based on the methods used to integrate tools, described in Chapter 2. There is not a quantitative scale for the rules; instead, if a solution is preferred, the term "prefer" is used. Applying the design rules is the second step of the process. The first step, determining the extendability class for the pair of tools, is described in Section 3.2.2.1. That step examines the extendability of the tool pair and places it in an extendability class.

The design rules use the values given to each of the functional dimensions, for both a single tool and the tool pair, to determine the structural dimensions for the integration. Design rules are applied in two phases. First, the communication path rules are applied, considering the input and output mechanism of the tool interface, to determine the communication path. The notation "X/Y" (for example, "stdout/stdin") is used to indicate *output of first tool/input of second tool* and is referred to as the "data interface" between the two tools. In the second phase, design rules are applied according to the extendability class to determine the control integration implementation, the data interface, and the data

transformation. In some cases, the extendability class rules note exceptions to the tool interface rules.

The next section describes the communication path rules, applicable to all extendability classes. The following sections contain design rules for each extendability class, in Table 3. Each of the sections that are specific to an extendability class include a diagram that denotes the control and data flow through the resultant system. Control flow in the system is indicated on the diagram by dashed lines, while data flow is indicated by solid lines.

Table 6 Communication Path Design Rules

Rule Number	**Tool Interface:**	**Communication Path:**
T1	file/(anything) or (anything)/file	Shared Data
T2	stdout/stdin	Stdin and Stdout
T3	Msg Passing/(anything) or (anything)/Msg Passing	Message Passing

3.4.1 Communication Path Design Rules. The communication path rules are based upon the interface of the tool pair and provide guidance regardless of the extendability class. In some cases, rules specific to the extendability class may contradict these rules. These rules address the communication path (Table 5), one of the structural dimensions of the integrated system based on the tool values defined in Table 2. Table 6 summarizes these rules.

- Rule T1: If file/(anything) or (anything)/file, the shared data communication path should be used, since files are easily used as shared data. Files can be made to allow access to both tools and can be manipulated into the desired format.
- Rule T2: If stdout/stdin, stdin and stdout should be used for the communication path since they are operating system standard input and output mechanisms. Stdin and stdout are inherently good communication mechanisms and this characteristic should be taken advantage of when integrating tools that use them.

- Rule T3: If messages are used to pass data, message passing should be used as the communication path. Tools that use messages to communicate can be linked to another tool by using message passing to communicate between them.

3.4.2 Neither Extendable. The neither extendable class should use middleware to handle the integration. Since the middleware is a centralized application, centralized control should be implemented. The nature of the tools dictates the implementation chosen for the central controller. If one of the tools is interactive and the other submits batch jobs, it is desirable to gather all of the necessary information from the interactive tool and use it to submit the batch jobs. Alternatively, if the input to the interactive tool can be stored and supplied to the tool without user interaction, that may be preferred. These decisions are based on the specific tools being integrated and their relationship to the process they support. The key is that centralized control can be used to customize the default method of input and output so that it is what the customer desires.

If one tool places its data in a central location and the other tool uses passed data, the middleware should perform transformations on the data so that both tools use shared data. The tool that depends on passed data cannot be changed, but the middleware can reroute the data so that the input source or output destination for the tool using passed data does not change. Since neither tool can be extended, the middleware must perform any necessary data conversions between the tools. Therefore, the data transformation value would be *Transformation via Middleware.*

The resulting system of a tool pair in this class would use middleware to execute the first tool, execute any data conversion routines and then execute the second tool. In this manner, data integration is achieved by executing the conversion routine and control integration is executed by providing a method for the user to invoke one program that in turn controls the other two tools, illustrated in Figure 2.

3.4.3 First Extend. Tool pairs that are in the extendability class *First Extend* have a set of design rules that are specific to the class to guide the integration effort. These

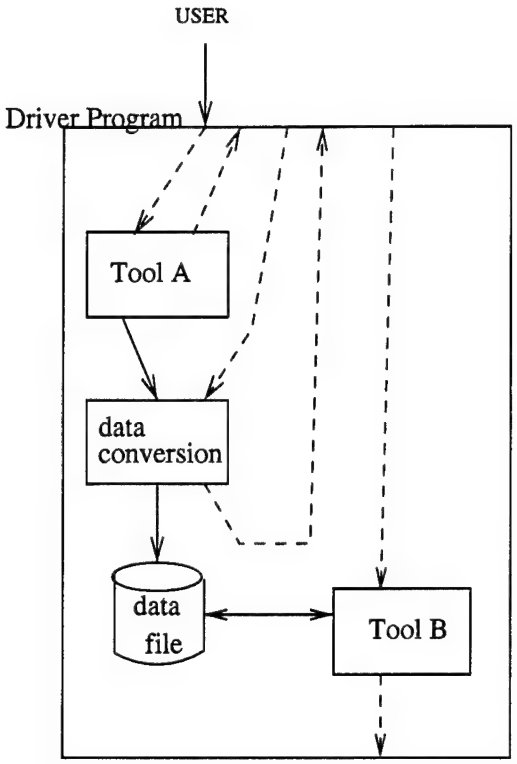


Figure 2 Neither Tool Extendable

rules are primarily based on the data interface between the two tools and provide guidance on the data compatibility of the integrated tools.

There are three general rules that are applied to this class, regardless of the tool interface. The first rule addresses control integration implementation. Since only the first tool of the pair is extended, the appropriate control integration implementation is client-server control. Client-server control is used when one tool controls the other, or executes functions in the other tool. Second, in this class, data transformation is performed by the output tool since the first tool can be extended to properly format the data.

The third rule describes reasons to change the extendability class of the tool pair. The integration of a tool pair in this extendability class may also be approached by following the rules for *Neither Extendable*. This solution may be desirable in the following cases: if the data conversion involves two data formats which are completely dissimilar, but could be transformed to a central format; if the extendability interface of the tool does not provide the functionality needed to completely integrate the two tools; or if the

extendability interface of a tool is not well-documented or well supported, making a driver program more maintainable. If this approach is chosen, the appropriate control integration implementation should be used, as described in the design rules for *Neither Extendable.*

The remaining rules for this class are presented in Table 7 and are described in more detail in Appendix D of this document. The letter/number combinations in the table refer to the designator for the related transformation, described in detail below the table.

Table 7 Design Rules for *First Extend*

	Stdin	Message Passing	File	Command Line
Stdout	F1	F5	F2	F2 or F3
Message Passing	F2	F5	F2	F2 or F3
File	F4	F5	F2	F2 or F3
Built-In Output	F2	F5	F2	F2 or F3

- Transform F1: Extend the first tool to convert the data to the proper format for the second tool. Develop a driver program to pipe the two tools together, using stdout and stdin to pass the data from the first tool to the second.
- Transform F2: Modify the first tool, if necessary, to write its data to a file. After ensuring this file is in the proper format for the second tool, execute the second tool.
- Transform F3: If the command line input of the second tool expects actual data on the command line, extend the first tool to build the command and execute the second tool.
- Transform F4: Extend the first tool to format the data properly for the second tool and build the command line for the execution of the second tool. Redirect the input from the file to the second tool as part of the command.
- Transform F5: Extend the first tool to gather all of its output data, ensure it's in the proper format for the second tool, and then execute the second tool. Pass messages to the tool from the data gathered from the first tool.

Data integration for a pair of tools in this class is accomplished by writing a data conversion routine to change the format of the data, location of the data, or both. Data integration is handled by a conversion routine executed by the tool that is extended. The

control integration is also achieved by the extended tool executing the tools in the proper order. The resultant architecture of the integrated system, illustrated in Figure 3, is straightforward.

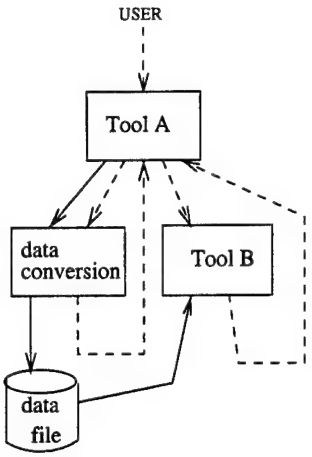


Figure 3 First Tool Extendable

3.4.4 Second Extend. The extendability class *Second Extend* also has a set of design rules that are specific to the class to guide the integration effort. In general, integration efforts in this solution class should use client-server control. The reasoning is similar to that given for *First Extend*; only one tool is extended, so that tool will be responsible for controlling the other tool and that meets the description of client-server control. For tool pairs in this class, data transformation is performed by the input tool since the second tool can be extended to properly format the data. Also, for the same reasons expressed in *First Extend*, it may be desirable to change the method of integration to follow the rules for the *Neither Extendable* class.

The remaining rules for this class are presented in Table 8 and are described in more detail in Appendix D of this document. The letter/number combinations in the table reference transforms, described below the table.

- Transform S1: Extend the second tool to convert the data to the proper format from the first tool. Develop a driver program to pipe the two tools together, using stdout and stdin to pass the data from the first tool to the second.

Table 8 Design Rules for *Second Extend*

	Stdin	Message Passing	File	Command Line
Stdout	S1	S2	S3	S3 or S4
Message Passing	S3	S2	S3	S3 or S4
File	S3	S2	S3	S3 or S4
Built-In Output	S3	S2	S3	S3 or S4

- Transform S2: Extend the second tool to gather all of the output from the first tool. Perform any necessary data conversion and then execute the second tool. Pass messages to the tool from the data gathered from the first tool.
- Transform S3: Extend the second tool to execute the first tool, save the data in a file, and perform any necessary data conversions. Use the data as input to the second tool.
- Transform S4: If the command line input of the second tool expects actual data on the command line, extend it to execute the first tool, build the command and execute the appropriate functions of the second tool.

In the resulting system, the second tool will execute the first tool when it is executed, perform any changes to the data that are necessary, and finally execute the functionality of the second tool that is desired. Data integration is handled by a conversion routine executed by the tool that is extended. The control integration is also achieved by the extended tool executing the tools in the proper order. Since extending the second tool is similar to extending the first tool, it is not illustrated.

3.4.5 Both Extend. Even though it is possible to extend both tools in this pair, it is preferred to choose one to extend, based on its control and data characteristics. Since it does not matter which tool is chosen, the tool that most lends itself to extension should be chosen. Extending two tools complicates the initial development as well as the maintenance of the system by increasing the complexity of the system. However, if one tool lends itself to data integration, while the other lends itself to control integration, both tools may be extended. If both tools have a GUI, the integrator should develop a driver program that will give a unified feeling to the system. In this case, centralized control is used. By

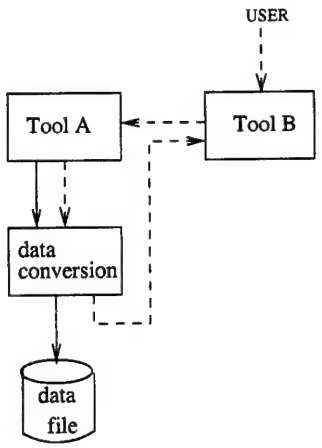


Figure 4 Both Tools Extendable

choosing one tool for extension, the design rules for *First Extend* or *Second Extend* are applicable.

If the decision is made to extend both tools, distributed control should be implemented. Using distributed control allows each tool to initiate functions in the other tool and retrieve the results, creating a distributed environment in the system. There are several possible methods for handling data and control integration for the tool pair. One tool may handle data integration while the other tool handles control integration. Alternatively, both tools could be extended to handle data integration, while only one implements control integration. Data integration may involve each tool pre- or post-processing the data. Additionally, a message passing scheme may need to be established to ensure the data is not in use by both tools simultaneously. For control integration, it is usually best to extend the first tool in the sequence to call the second tool, allowing the user to think logically about the progression of the integrated system. However, in a case where the first tool in the sequence does not lend itself to control integration, the second tool may also be used for that aspect of integration. An example of the resultant architecture of extending one tool for data and one for control is illustrated in Figure 4.

3.5 Two-Way Communication

In some cases, the tool pair being integrated may need to have two-way communication, that is, communication from each tool to the other tool. In this case, tool integration should be approached, using this methodology, as if the tools are two pairs of tools, one with communication in one direction and one with communication in the other direction. By approaching the integration in this manner, the integration is broken down into two smaller pieces and the methodology presented in this chapter may be used to guide the integration. As the methodology is applied, it is possible that it will recommend extending one tool for each integration, causing both tools to be extended for the overall integration. If this is the case, the rules for ***Both Extend*** should be examined and, if possible, only one tool should be extended. Otherwise, the methodology should be followed in the same manner as integrating a tool pair with one-way communication.

3.6 Summary

This chapter provides an overview of the methodology developed as part of this research effort. Two concepts were combined to develop this methodology: the concept of a design space, composed of functional dimensions, structural dimensions, and design rules, and the concept of integration classes. The step-by-step approach described here is illustrated in the next chapter through the application of the methodology to integrations involving *AFIT*tool.

IV. Application of Methodology to AFIT*tool*

The integration of tools can be accomplished in several ways, some of which were described in previous chapters. *AFIT*tool was integrated with three other tools: a parser for architectural specifications written in Acme [12], Rational Rose 98, a CASE tool for software development [21], and daVinci, a graph layout tool [10].

The integration of *AFIT*tool was accomplished using two types of integration from Wasserman's model, namely control and data integration. Platform integration is ignored since all of the tools reside on the Unix platform, making platform integration unnecessary. Presentation integration is considered in the criteria for the tools to be integrated, by considering the interface to the tools. However, it is not the primary concern for this integration effort; functionality of the resulting toolset is the primary concern. Process integration, how the tools fit into the software process, is another type of integration that is considered in the selection of tools to integrate and is not used during the integration effort. The next sections contain detailed descriptions of the integrations implemented between *AFIT*tool and the tools listed above. The final section of this chapter describes how the methodology described in the previous chapter meets the requirements of control and data integration, as developed by Thomas and Nejmeh.

4.1 Integration of AFIT*tool*

To solve the problem described in Chapter 1, *AFIT*tool was integrated with Rose, the Acme parser, and daVinci. Each tool chosen met all of the criteria for tool integration and addressed at least one shortcoming of *AFIT*tool, described in Chapter 2. All three tools improved the user's ability to analyze the model by offering different views of it. In addition, the integration with Rose offers another method of inputting domain models, with an interface that is more user friendly than the current *AFIT*tool interface.

This integrated system is illustrated in Figure 5. Rose is used to develop the informal model of the software system, including class specifications and state diagrams. The user is able to define other diagrams in Rose that may help with general understanding, but these are not used by *AFIT*tool. Rose provides the user the capability of entering textual

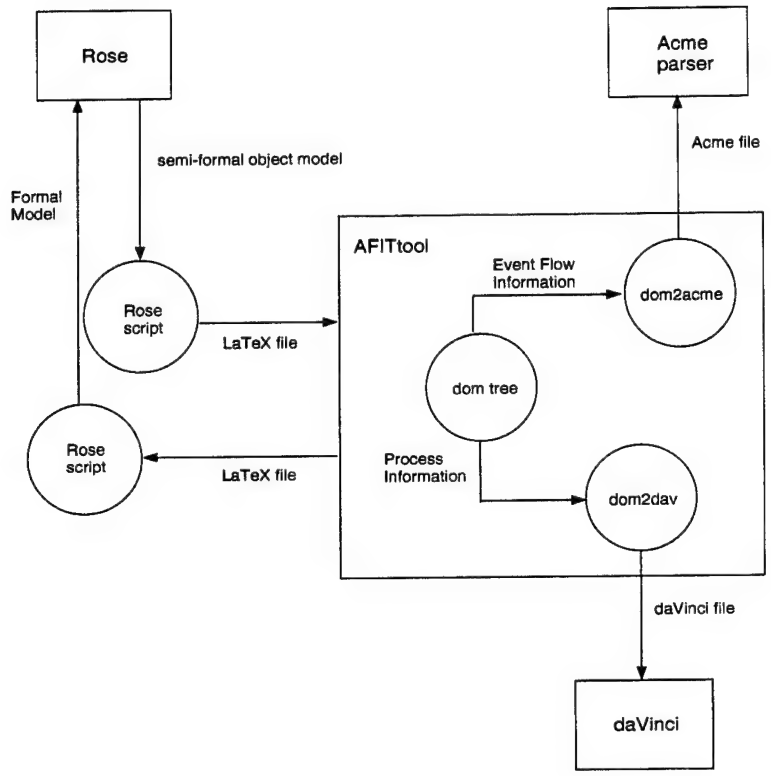


Figure 5 Overall System Integration Concept

information which can be used for formal constraints and this information is used in the integration. Through the use of a Rose script, the information needed to populate the domain model in *AFIT*tool is gathered from the Rose diagrams and output to a LaTeX file that can be parsed into *AFIT*tool using its current capabilities.

Rose can also be used to develop object model and state diagrams of a domain from an existing file. Through the use of another Rose script, a LaTeX file in the format *AFIT*tool expects can be used as input to create Rose diagrams. This capability can be used to create object and state diagrams from existing domains, with little help from the user.

Acme extends the capability of *AFIT*tool by allowing the user to generate an architecture specification for the system. The class and event flow information, stored in the domain model, are used to generate an architecture based on the hierarchical object model.

*AFIT*tool was integrated with daVinci in order to display the process diagram of the currently loaded domain model. Processes are represented by circles and data flows are represented by lines going from one circle to another, with the name(s) of the data element(s) on the line. The process diagram generated from the model can assist the user in visually checking the correctness of the data flow model.

4.2 Integration of AFIT*tool and the Acme parser*

The Acme parser is a batch tool with a command line interface. By default, the parser uses standard in (stdin) and standard out (stdout) for input and output. The source code for the parser is available, allowing it to be extended for control or data integration purposes. By gathering information from the domain model in *AFIT*tool, Acme code is generated representing the object structural model. The Acme file is then sent to the Acme parser for syntax checking and re-formatting. The Acme parser generates output that is re-formatted according to the approved format of an Acme file. This integration was accomplished by following the methodology presented in Chapter 3. The following paragraphs describe the steps taken.

4.2.1 Representing the Domain Model in Acme. The Acme language provides a method of representing the architecture of a system. The domain model stored in *AFIT*tool can be represented in Acme in a hierarchical fashion. For instance, the relationships between classes that are stored in the domain AST, such as aggregation and inheritance, can be represented in the Acme language. If there is only one level in the domain, such as a single primitive component, that can also be represented in Acme.

In this integration, the event flows were chosen to represent the interaction between *classes* in the domain. The state transition table, part of the LaTeX file used as input to *AFIT*tool, contains the information needed for generating the Acme file of event flows. Table 9 is an example of a state transition table for the SubCounter class. Figure 6 contains the output generated from Acme as well as the corresponding event flow diagram. The diagram itself is not created by the tool, but could be drawn by the user to check the

Table 9 State Transition Table for SubCounter Class

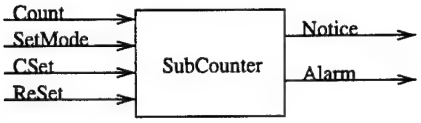
Current	Event	Guard	Next	Action	Send
CountingUp	ReSet		CountingUp	ResetCount	
CountingUp	CSet		CountingUp	SetCount	
CountingDown	CSet		CountingDown	SetCount	
CountingDown	ReSet		CountingDown	ResetCount	
CountingUp	SetMode	$newmode = down$	CountingDown	SetModeDown	
CountingUp	Count	$thecount < limit$	CountingUp	IncrementCount	
CountingUp	Count	$thecount = limit$	CountingUp		Alarm
CountingDown	SetMode	$newmode = up$	CountingUp	SetModeUp	
CountingDown	Count	$thecount > 0$	CountingDown	Decrement	
CountingDown	Count	$thecount = 0$	CountingDown		Alarm
NotReset	Count	$thecount > 0$	NotReset	Decrement	
NotReset	Reset		Reset		Notice
Reset	Count	$thecount > 0$	Reset	Decrement	

validity of the system. An example of the Acme representation of an aggregate class is in Appendix E of this document.

The components of the architecture model are the classes of the object model, while the connectors are the event flows to and from these classes. The end result is an architectural description of an event flow diagram. A distinction is made between aggregate and primitive classes in several ways. An aggregate class is of type AggregateClass in Acme, and is developed with a Representation containing the necessary information on the classes that make up the aggregate. These classes can be either primitive or aggregate classes, and the architectural description represents multiple level aggregates if they exist in the domain.

Primitive classes, those classes that are not composed of other classes, are represented architecturally with ports, connectors, and attachments between the connectors and the ports. They are of type PrimitiveClass. Each port is assigned to a component and is of type SendPort, if the event is sent to a destination outside the class, or ReceivePort, if the event is received from outside the class. Each event is assigned a connector, of type EventFlow, as a path for the event to travel along. The connector has a destination and a source, each of which are attached to a port.

The interactions between classes, based on event flows in the domain model, are represented by placing connectors between the classes. Each class must have ports for




```
System c3 : ObjectEvent = {
 Component Counter : PrimitiveClass;
 Component SubCounter : PrimitiveClass = {
  Port Alarm : SendPort;
  Port CSet : ReceivePort;
  Port Count : ReceivePort;
  Port Notice : SendPort;
  Port ReSet : ReceivePort;
  Port SetMode : ReceivePort;
 };
 Connector AlarmEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
 };
 Connector CSetEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
 };
 Connector CountEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
 };
 Connector NoticeEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
 };
 Connector ReSetEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
 };
 Connector SetModeEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
 };
 Attachments {
  SubCounter.Notice to NoticeEvent.source;
  SubCounter.Alarm to AlarmEvent.source;
  SubCounter.Count to CountEvent.sink;
  SubCounter.SetMode to SetModeEvent.sink;
  SubCounter.CSet to CSetEvent.sink;
  SubCounter.ReSet to ReSetEvent.sink;
 };
```

Figure 6 Output of the Acme parser

each *event type* sent or received. Attachments are made between each event flow connector and the correct port, based on which class sends and which class receives the event type. Through this mapping, the event flows in the system are represented by connectors between classes and ports on classes. In Figure 6, one example of an event type is the connector *AlarmEvent*. The port *Notice* has both a role named *sink* and a role named *source*, identifying each end of the connector.

Each class in the object model is represented by a *component* in the architecture. Primitive classes, of type PrimitiveClass, are represented in Acme by defining the event types valid for that class. Each class has a port for each event type sent or received. If the class is the originator of the event type, the port is of type SendPort, and if the event type is received from outside of the class the port is of type ReceivePort. Notice that in Figure 6 the component SubCounter has a port for each event type in the class. Connectors are attached to ports of components, representing an event flow between the components. Each event type is assigned a connector, of type EventFlow, as a path for the event to travel along. The connector has a destination connected to a receive port and a source connected to a send port. When the domain model is processed, connectors are made for each event type for each class. Attachments are detected between two classes if two classes in the domain use the same name for an event type. This name matching provides the sender and receiver of the event type, allowing each Acme connector to be attached to one SendPort in the sending class and one ReceivePort in the receiving class. In some domains, however, both the sender and receiver may not be present, leaving a dangling event flow.

Each aggregate class is of type AggregateClass in Acme, and is developed with a Representation containing the necessary information on the classes that make up the aggregate. These classes can be either primitive or aggregate classes, and the architectural description can represent multiple level aggregates, if applicable. Aggregate classes also have ports, connectors, and attachments for the events flowing at the level of the aggregate class. For each primitive class that is part of an aggregate class, the information described above is included. For each aggregate class that is part of another aggregate class, information is included on each component of the aggregate.

The resulting architecture description can be used to visually check the validity of the event flows of the system, ensuring every event type has a sending class and a receiving class, denoted in the Attachments section of the Acme code. If the sender or receiver class of an event type is not in the domain, the user can enter that information into *AFIT*tool and generate the Acme code again. The parsed Acme code can also be annotated, using a text editor, for use in other tools, such as Rapide and Wright. Rapide can be used to simulate the system, while Wright can be used to translate the code into other architecture languages. Information on how to do this can be found in the Wright and Rapide documentation [18] [17].

4.2.2 Application of the Methodology. As described in the methodology, the first step is to determine the input, output, and extendability of each tool. The input and output values for the Acme parser are *Stdin* and *Stdout*. For *AFIT*tool, the input and output values are *File* and *Built-in Output*. The extendability value for both tools is *Source Code Available*. The next step is to determine the extendability class and data compatibility for the tool pair. In this case, the extendability class is *Both Extendable* and the data compatibility value is *Not Compatible* since the data in *AFIT*tool is not in the correct format for the Acme parser.

From these values along the functional dimensions, the design rules are applied to determine the values for the structural dimensions. Since the Acme parser was extended to take file input, tool interface rule T1 applies, and *Shared Data* is determined as the value for the communication path. By applying the design rules for *Both Extend*, it was determined that only one tool should be extended, even though it is possible to extend both tools. The decision was made, however, to perform minor extensions to the Acme parser to allow file input and output. This decision follows the spirit of the design rule, and allows a cleaner interface between the two tools. Therefore, the extendability class is changed to *First Extend*, since *AFIT*tool contains the information needed by the Acme parser.

Next, the rules for *First Extend* were applied since the majority of the extension was performed to *AFIT*tool. Following the guidance on control integration, the value for

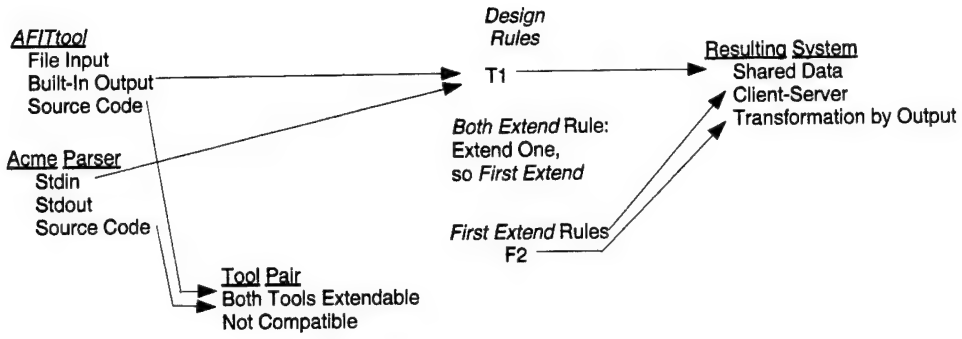


Figure 7 *AFIT*tool/Acme Parser Integration

the control integration dimension was determined to be *Client-Server.* Transform F2 was used, directing that the built-in output of *AFIT*tool be captured in a file in the proper format for the Acme parser. By applying this rule, the value of the structural dimension data transformation is *Transformation by Output Tool.* The values for the functional and structural dimensions of this integration are summarized in Figure 7. The next two subsections describe in detail how the integration was implemented, based on the decisions made by following the methodology.

4.2.3 Data Integration. Data integration between *AFIT*tool and the Acme parser was accomplished by extending both tools. The Acme parser was extended to take input and output file names on the command line. Both the input file name and output file name are provided by the user. *AFIT*tool was extended to generate an Acme file, in the syntax expected by the parser. Refine code was written to read the domain Abstract Syntax Tree (AST) in *AFIT*tool and write an output file containing the architectural information for the Acme source file. To generate this file, each class in the domain is examined. First, aggregate classes are examined, generating event types for any events sent or received at the aggregate level. Next, each primitive class that is part of an aggregate is examined and Acme code is generated to represent the aggregate as a composition of its primitive classes. Each primitive class is examined to determine any event flows sent or received by the class, and an Acme connection is generated to represent the event type. Finally, primitive classes that are not part of an aggregate class are examined and processed, generating event flows

for each class. In addition to the event flows, each class has a port for each event type and an attachment between each event flow and its respective port.

4.2.4 Control Integration. Control integration of the Acme parser and *AFIT*tool was accomplished by extending *AFIT*tool to generate the Acme file and to invoke the Acme parser. The extensions to *AFIT*tool were accomplished through the use of Refine, both the language and the environment of *AFIT*tool. The user is able to select the generation of Acme code from the domain menu of *AFIT*tool. When this option is selected, the data transformation program that was written to gather information from the domain model and output it in Acme is invoked. The program also calls the Acme parser, passing the input and output file names received from the user. The Acme parser then executes and prints an error code in the *AFIT*tool window if it detects an error or a "TRUE" if it achieves successful completion. After the parser executes, control is returned to *AFIT*tool and the user may choose another option from the menu. If the parser detects an error, it is reported to the user, allowing the revision of the domain model. Any errors that are reported by the parser are not handled by *AFIT*tool in any way. After any revisions, the Acme code must be generated and parsed again.

4.3 Integration of Rational Rose 98 and AFIT*tool*

Rational Rose 98 is a CASE tool, designed to be used throughout the life cycle of a software system. Rose, however, uses semi-formal methods to specify the systems, while *AFIT*tool uses formal methods, ensuring the final system is correct with respect to the requirements specification. By combining these two tools, the user is able to informally specify the domain of the software system and to embed formal constraints in Rose's textual fields, send the information to *AFIT*tool, and use *AFIT*tool's formal capabilities to complete the design transformations and code development of the specification. The integration of Rose and *AFIT*tool was separated into two integrations, one from Rose to *AFIT*tool and the second from *AFIT*tool to Rose. The next sections describe the first integration.

4.3.1 Representing Rose drawings in the Domain Model. The first step in allowing Rose drawings as input to *AFIT*tool is to specify the system, through drawings, in a manner that *AFIT*tool understands. The current input to *AFIT*tool is a *Z* specification in LaTeX format. This format is difficult to write and often difficult for the user to understand. Although the proper use of a formal language guarantees the code produced will correctly represent the specification, if the specification is incorrect the system will be incorrect. The use of a semi-formal tool increases the likelihood of a correct specification because the tool makes it easier for the user to understand what he or she has specified. Rose specifications are essentially a series of drawings, meaning the user has to understand the syntax of the drawings, a task that is often easier than understanding a formal language such as *Z*. Because the user understands the drawings, sometimes he or she is able to detect flaws they would not detect by looking at the LaTeX *Z* specification. By combining Rose and *AFIT*tool, the user is able to use semi-formal methods to specify the domain of the software system, send the information to *AFIT*tool, and use *AFIT*tool's formal capabilities to complete the design transformations and code development of the specification.

The Rose drawings that are used to develop the *AFIT*tool domain model are class diagrams and state models. The other diagrams the user develops in Rose are not considered in the transformation from Rose to *AFIT*tool. This transformation process expects the diagrams to be in a specified format, compatible with *AFIT*tool. For example, some symbols need to be specified in LaTeX *Z* in order for the transformation to work correctly. The mapping from Rose to *AFIT*tool is described in detail in Appendix C, including the proper format for each field in the Rose diagram.

4.3.2 Application of the Methodology. The first step in integrating the two tools is to determine the input, output and extendability values for each of the tools. *AFIT*tool uses *File* input and produces *Built-In Output*. Rose uses *File* input and output. Both tools are extendable, Rose via *Tool Provides Extendability* and *AFIT*tool through *Source Code available*. Next, the extendability class is determined from the combination of the two tools. In this case, the extendability class is *Both Extendable*. The value of the data

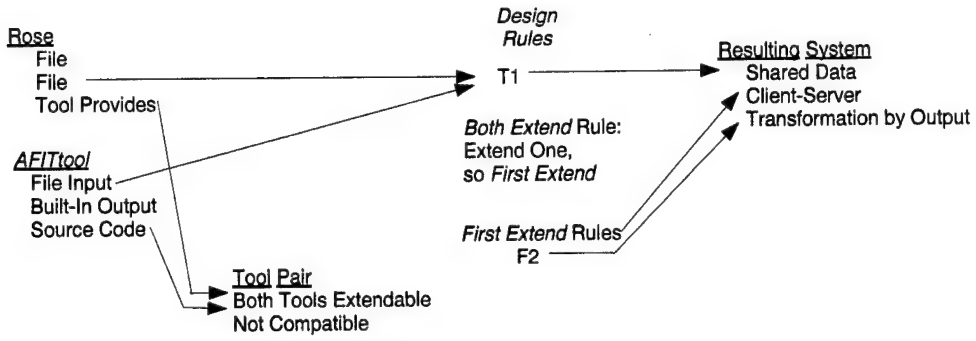


Figure 8 Rose/*AFIT*tool Integration

compatibility functional dimension is ***Not Compatible*** since the data in Rose is not in the format required by *AFIT*tool.

The third step is to apply the design rules, supplying the functional dimensions as input. First, communication path rule T1 was applied, since the Rose output mechanism is ***File***, making the communication path value ***Shared Data***. Next, design rules for ***Both Extend*** were applied, recommending that only one tool be extended. In this case, the decision was made to extend *AFIT*tool for control purposes only and Rose for data purposes, causing the extendability class to change to ***First Extend***. The decision to extend *AFIT*tool for control was made because *AFIT*tool's interface does not lend itself to being controlled by another tool. Based on the design rules for ***First Extend***, the value for control integration implementation was determined to be ***Client-Server*** and the value of data transformation is ***Transformation by Output Tool***. Transform F2 was used, since both tools use files. The values for the functional and structural dimensions are summarized in Figure 8. The next two sections describe in detail how the integration was implemented, based on the decisions made by following the methodology.

4.3.3 Data Integration. Data integration of Rose and *AFIT*tool was accomplished by using the Rose Extensibility Interface, the provided method for extending Rose [29]. Included in the interface is a scripting language, Summit BasicScript, very similar to Visual Basic. Most of the information captured in Rose drawings is accessible from within scripts. The information gathered from the class specifications is written to separate LaTeX files, one for each class, named *classname*.tex, where *classname* is the corresponding class name.

These files are automatically generated by the Rose script when the option for "Output Model to LaTeX" is chosen from the Tools | AFITtool menu in Rose. These files are in the proper format for *AFIT*tool and can be parsed into an *AFIT*tool domain by the user.

*AFIT*tool requires Z schemas for each section of the definition of a domain. An example of the template used for the file parsed by *AFIT*tool can be found in Appendix A of this document. Each section of the file gives an informal definition and a formal definition of the current "piece" of the model. The file is broken into three main parts: Structural, Functional and Dynamic Models. The Structural Object Model includes an informal description of the object, including the name, date, and author, as well as its attributes and types. The formal section of the Structural Model consists of one or two Z schemas. The first is required, as it contains the attributes and any class constraints that must hold at all times. The second schema is an initialization schema. It may be omitted if the class does not have initialization values for its attributes. Associations detected in the Rose diagram are represented in the Structural Model of the class. If the diagram contains an associative object, it is represented in Z LaTeX by a separate class schema. The cardinality of associations in the model is not explicitly represented in the Structural Model, but is used to determine the proper function.

The Functional Model of the object includes a Z schema for each operation defined in the class. The name of the class must be included, either as Read-only (Ξ) or as Read-Write (Δ). For files automatically produced by Rose, the class is always included as Read-Write. Input parameters to the operation are decorated with a "?" and output parameters are decorated with a "!". Local variables included in the operation exist only in the scope of the operation and are not decorated.

The final section of the file is the Dynamic Model. This model includes the state model for the object class. A Z schema is defined for each state and each event in the model. Each state includes the class schema, through the schema inclusion mechanism of Z. Each event may include input parameters and constraints on those parameters. The dynamic model is summarized in the state transition table, showing the complete set of transitions for any object. The current state, next state, trigger event, guard condition(s), action(s), and send events are included in the table.

4.3.4 Control Integration. Since both tools are extendable, a decision was necessary to complete the integration of Rose and *AFIT*tool. The domain tool menu of *AFIT*tool was extended to include an option to create a domain model in Rose. When the user chooses this option, Rose is started and both Rose and *AFIT*tool will execute simultaneously. The user can then develop a domain model in Rose, including classes and state models. After outputting the domain to a LaTeX file by choosing the option "Output Model to LaTeX" from the Tools | AFITtool menu in Rose, the user can return to the *AFIT*tool window and load the domain, with Rose still running. If any corrections need to be made, the user is able to return to the Rose window, make the corrections, and generate the LaTeX file(s). Then the user may return to *AFIT*tool and load the class(es) again. Each time changes are made, this process is reproduced to pass the changes from Rose to *AFIT*tool. This allows the user to participate in an iterative process to develop the domain, developing the informal and formal models of the domain through a series of corrections.

4.4 Integration of AFIT*tool and Rational Rose 98*

In some cases, it is desirable to use the same input source for more than one tool. Although this possibility is not explicitly discussed in the methodology, it was demonstrated as part of this research. By enabling Rose to use *Z* LaTeX files to create Rose diagrams, both Rose and *AFIT*tool can use the same input. Although *AFIT*tool does not have to be executed in order for Rose to read the LaTeX file, the file must be parsable by *AFIT*tool.

4.4.1 Representing the Domain Model in Rose drawings. The LaTeX file is expected to be in the format of the template in Appendix A. The order of the file is important; if the file is not in the order expected by the Rose script, the proper diagrams will not be created. The Rose script expects each file to contain only one class, and recognizes a class, an initialization schema, zero, one or more events, zero, one or more states, and a state transition table. Additionally, if the file represents an aggregate class, the file may contain zero, one or more associations and/or associative classes. In the case of a domain that contains an aggregate class, it is necessary to first parse in the primitive classes and then the

aggregate class. If the primitive classes are not in place before the aggregate class is read, the diagrams will not be created correctly. Just as the cardinality of associations is not captured by the Rose to *AFIT*tool data conversion, the cardinality is also not generated from the domain model to the Rose diagrams.

4.4.2 Application of the Methodology. Although the methodology does not specifically discuss extending a tool to use the input generally used by another tool, the methodology can be applied by considering an integration between the tool that created the input, for example, the LaTeX file, and the tool that will use the same input file. In this demonstration, the tool that created the file, which does not need to be known, is the "first tool" and Rose is the "second tool." First, the functional dimensions for the first tool are determined. The input mechanism is not known, the output of the first tool is *File*, and the extendability is *Neither*. For this demonstration, the values for the functional dimensions of Rose are the same as for the first integration. For the tool pair, the extendability class is *First Extend* and the data compatibility value is *Not Compatible* since the LaTeX file is not in the proper format for Rose.

The next step is to apply the design rules, supplying the functional dimensions as input. Communication path rule T1 was applied again, since the "phantom" tool output mechanism in this case is the LaTeX *File* and the Rose input mechanism is *File*, making the communication path value *Shared Data*. Finally, design rules for *Second Extend* were applied to determine the data transformation and control integration implementation. The rules for *Second Extend* recommend the control integration implementation be *Client-Server* and the data transformation be *Transformation by Input Tool*. Transform S3 was used, since both tools use files, instructing that the second tool save the data from the first tool in a file and use it as input. The values for the functional and structural dimensions are summarized in Figure 9. The next two sections describe in detail how the integration was implemented, based on the decisions made by following the methodology.

4.4.2.1 Data Integration. In order to enable Rose to create diagrams for existing LaTeX models, data integration was accomplished by creating a Rose script. The Rose script is used to extend Rose by translating the data used by *AFIT*tool into a format

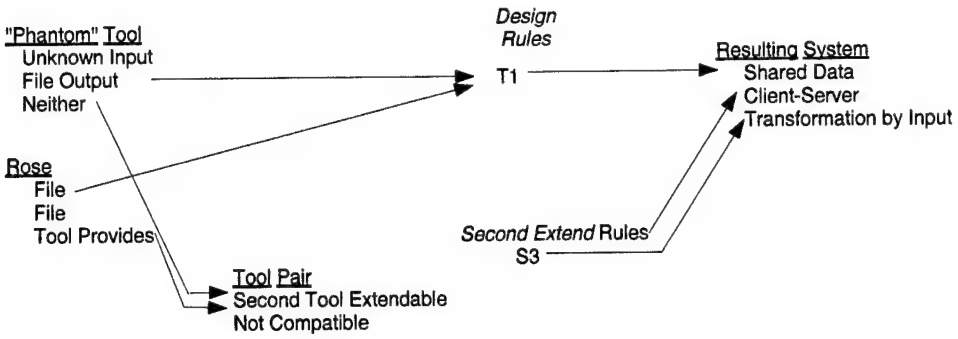


Figure 9 *AFIT*tool/Rose Integration

it can use. In this case, the script reads the LaTeX file and creates the Rose model simultaneously. Since the LaTeX file used by Rose to create the domain diagrams is the same one that is used by *AFIT*tool, it is suggested that the file be parsed into *AFIT*tool before it is parsed into Rose, to detect any errors in the LaTeX file syntax. After the file is parsed into Rose, semantic errors in the file can be detected through the inspection of the diagrams. The diagrams can then be changed and the LaTeX file can be re-created through the use of the script discussed in previous sections. In this manner, an iterative process can be achieved in which the domain is perfected.

4.4.2.2 Control Integration. Control integration in this demonstration is accomplished through the extension of the Rose menu. In order to use the LaTeX file as input, the user chooses the option "Read in LaTeX file" from the Tools | AFITtool menu in Rose. This selection causes a dialog box to appear, asking the user for a .tex filename. The default directory is the one from which Rose was started. The .tex file specified is parsed by Rose and the appropriate diagrams are created.

4.5 Integration of AFIT*tool and daVinci*

The third tool that was integrated with *AFIT*tool is daVinci, a graph layout tool developed at the University of Bremen, Germany. It was integrated with *AFIT*tool to provide a visualization of the data flows in the domain model. After a domain is loaded into *AFIT*tool, the user may choose to display the data flow diagram in daVinci. Each process is represented by a circle, and each data flow between processes is represented by

an arrow from one circle to another, with the name of the data displayed on the line. The user is able to visually check the data flow diagram and spot errors in the model which can then be corrected in the input file, or the Rose model if that mechanism was used, and re-loaded into *AFIT*tool.

4.5.1 Application of the Methodology. Applying the methodology in this integration indicates that the values for the functional dimensions are as follows. The input and output of daVinci are *File*, while the input and output of *AFIT*tool are *File* and *Built-in Output*. The extendability of *AFIT*tool is *Source Code Available* and for daVinci is *Tool Provides Extendability*, since daVinci allows an RPC interface to be established. From these values, the extendability class for this tool pair is *Both*. The data compatibility value is *Not Compatible* since the information in *AFIT*tool is not in the right format for daVinci.

The next step in the methodology is to apply the communication path rules. The communication path was determined to be *Shared Data*, by applying rule T1, since daVinci takes a file as input. Next, the design rules for *Both Extend* are applied and it is determined that only one tool should be extended. The decision was made to extend only *AFIT*tool, and use daVinci's capability to take an initialization file. The extendability class, therefore, changed to *First Extend*. By applying the design rules for *First Extend*, the control integration implementation was determined to be *Client-Server* and the value of data transformation is *Transformation by Output Tool*. Transform F2 applies to this integration and instructs that the built-in output of *AFIT*tool be captured in a file, in the proper format for daVinci. The values for the functional and structural dimensions are summarized in Figure 10.

4.5.2 Data Integration. In order to integrate *AFIT*tool with daVinci, *AFIT*tool was extended to output a file containing information for daVinci to build a data flow diagram. Since daVinci is a graph layout toolkit, it does not require any information on placement of the nodes or edges. In addition, daVinci makes provision for extension with its Application Programmer Interface (API) commands. There are two options for using these commands: write an interactive driver program which creates a separate daVinci process

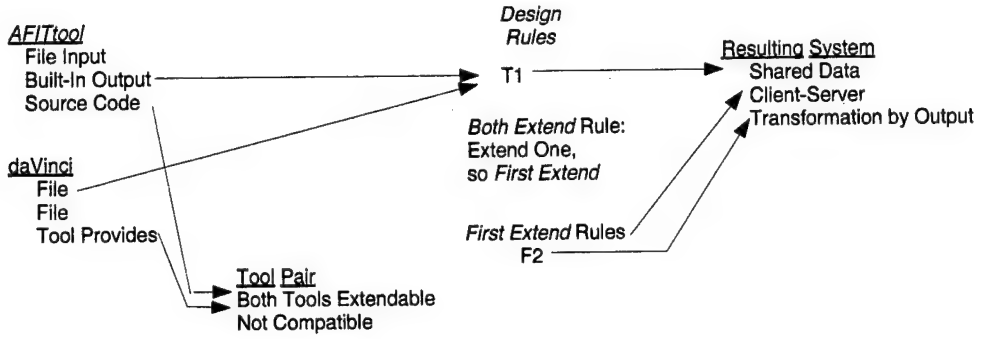


Figure 10 *AFIT*tool/daVinci Integration

or create a file that consists of daVinci API commands and include the file name on the command line, i.e. **daVinci -init command_file**. With the first option, commands are sent via a pipe to daVinci and responses are received by the program. The program, therefore, is responsible for handling the communication flow between the program and daVinci. The program must handle all possible responses from daVinci. This is the desired approach for interactive applications. The second approach was chosen for this integration effort, though, since daVinci is being used to display the data flow diagram and does not need to allow changes.

Data integration of daVinci and *AFIT*tool was accomplished by extending *AFIT*tool to create a file with the necessary graph commands in it. This file is created in several steps, the first of which is to examine all classes in the domain model, focusing on the processes in each class. Information on the processes is contained in the **has-operations** for each aggregate and primitive class in the domain. Next, every possible pair of processes in each class is examined to determine if the input parameter of one process is the output parameter of another class. Processes are linked based on name matching of the parameters. Finally, information for each process is written to the daVinci file. If a process shares data with another process, i.e. two processes have a parameter by the same name, commands are written to the daVinci file to create a node for each process and an edge between them with the name of the parameter on it. If a process is determined to be self-contained, not sharing data with another process, the daVinci command to create a node for the process is written to the file. The file is named *classname*.daVinci, where *classname* is the name of the class.

4.5.3 Control Integration. Control integration of *AFIT*tool and daVinci was accomplished by extending the *AFIT*tool domain menu. The user executes daVinci by choosing the daVinci option from the *AFIT*tool domain menu. After the user chooses this option, a separate data file is created for each class in the domain. Each data file, as explained above, contains API commands in the proper syntax for daVinci to build the requested graph, representing the data flow diagram for the class.

To draw the graph indicated by the saved file, daVinci is executed by *AFIT*tool with the `init` option activated and the filename specified, i.e. `daVinci -init classname.daVinci`. This option causes daVinci to read the file and execute the API commands when daVinci is started, drawing the data flow diagram. In this integration, the last command in the file saves the graph as *classname*.graph . By saving the graph, the user is able to run daVinci at a later time and open the graph without re-building it.

When daVinci is executed, separate daVinci windows, each with its own process identifier, are opened for each class. Each daVinci window contains the specified data flow diagram for a class. By starting a new process for each class, all of the data flow diagrams in the domain are displayed simultaneously. This allows the user to look at the process diagrams while making changes to the domain model. The user is responsible for closing the windows when they are no longer needed. If changes are made to the *AFIT*tool domain model, they will not be reflected in the graph file until it is rebuilt. To rebuild the graph file, the user must select the daVinci option from the *AFIT*tool menu again. Choosing this option causes the daVinci files to be re-generated and the diagrams to be displayed, allowing the user to repeat the process as necessary.

4.6 Validation of the Integration Methodology

This chapter has described in detail how the methodology developed as part of this research was applied to the integrations accomplished with *AFIT*tool. For each integration, the steps taken to set up the integration and the actual methods used to accomplish data and control integration, two classes in Wasserman's integration model, were described. Thomas and Nejmeh identified five properties of data integration and two properties of control integration in their examination of Wasserman's model, explained fully in Chap-

ter 2 [35]. This section explains how these properties relate to the methodology and the integration demonstrations presented here.

1. Data Integration Properties

   - Interoperability: Data interoperability between the tools is addressed by this integration methodology by ensuring the data is in the correct location and format for both tools. In some cases, getting the data to the right location and format involves some sort of data transformation, while in other cases, the tools agree on format and location before integration. In the case of writing a separate driver program, data transformation is accomplished by a routine in the program to write the data in the proper location and format for all tools. When one or both tools are extended, the extensions must handle the necessary data transformations.
   - Nonredundancy: Nonredundancy is accomplished in this methodology by ensuring that the tools use central data when possible. Central data, as described earlier, can be in the form of common objects, shared files or a common database.
   - Data consistency: The consistency of data is important when two tools use the same data, as is the goal in tool integration. The methodology ensures the data is consistent in two ways. First, data integration ensures only one tool is writing to the data at any given time. Second, control integration accomplishes intertool communication when data is shared, so that each tool reports when it has completed its changes.
   - Data exchange: Data exchange, succinctly, is the practice of two tools needing to pass or share data. In tool integration, the tools must agree on semantics and format for the data. The proposed methodology for data integration achieves proper data exchange by ensuring that either the driver program, if no tools are extended, or the tools themselves, if one or more tools are extended, perform any necessary data transformations so the data meets the agreed upon format and semantics.

- Synchronization: The property of synchronization is primarily concerned with the consistency of nonpersistent data, requiring each tool to report any changes made to the data to the other tool. Synchronization is accomplished in this methodology by control integration, adding some sort of reporting scheme in the driver program or to an extended tool.

2. Control Integration Properties

   - Provision: In the proposed methodology, only tools that have semantic agreement on data are integrated, having the effect of satisfying the provision property of control integration. As provision requires that any tool integrated into the environment is used, ensuring semantic compatibility meets this requirement. Although it is possible that the tool being integrated will make sense in the environment yet not be used, that is not in the spirit of the methodology as it is proposed.

   - Use: Use is highly related to the provision property in that it is a measure of the extent to which the tool is used in the integrated environment. Again, the spirit of the methodology is that only useful, semantically compatible tools will be integrated in the environment. The control integration guidelines ensure the availability of the functionality provided by each tool, causing the requirements of the use property to be met if the tools chosen for integration make sense together.

*AFIT*tool was integrated with three tools, the details of which were presented in this chapter. The information given demonstrates how to apply the methodology as well as how the methodology supports ideas presented by other researchers. The next chapter summarizes this thesis effort.

V. *Results, Conclusions and Recommendations*

Although this research effort began with the primary objective of enhancing *AFIT*tool to reduce its shortcomings, it very quickly became an effort to develop a generic methodology for integrating tools. The increased use of commercial off-the-shelf (COTS) software and government off-the-shelf (GOTS) software has driven some companies and government agencies to use several software tools to accomplish their mission. Using several tools is often more difficult and confusing than using one, so some software developers have attempted integrating these off-the-shelf tools to form a cohesive tool capable of accomplishing the mission.

After a thorough search of the literature, it was determined that there is not a generally accepted standard approach for integrating software tools. There are several models that offer a picture of the resulting integrated system, and upon which integration can be based, but there was not a step-by-step approach to use during integration. The next section of this chapter summarizes the work accomplished during this research effort. The following section analyzes the impact of one sample integration. The final section discusses recommendations for future work, both for improving the methodology and for improving *AFIT*tool.

5.1 Results

The generic methodology developed during this research was partially based on the concepts of two groups of researchers: environment integrators and software architects. Anthony Wasserman's model for tool integration was used as a basis for the two types of integration covered by the methodology [36]. The characteristics for each integration type, developed by Thomas and Nejmeh, helped to further define the integration types [35]. Software architecture played a large role in the development of a design space, complete with structural and functional dimensions and design rules, concepts presented by Lane [25].

Wasserman's original model of integration has five classes of tool integration: platform, presentation, data, control, and process integration [36]. For this research, only data and control integration were considered due to the nature of the integration effort.

Data integration is concerned with ensuring data used in the integrated system is in the proper form for any component that needs it. Control integration focuses on maintaining necessary communication between tools after they are integrated. Since the primary goal was to integrate *AFIT*tool with tools that could improve its shortcomings, presentation, platform and process integration were not directly addressed. Presentation integration focuses on how the integrated system looks to the user. This was addressed in the criteria for choosing tools to be integrated with *AFIT*tool and did not need to be addressed separately. Platform integration was not considered directly because it was also addressed in the tool criteria. One of the primary criteria was that all of the tools reside on the Unix platform, making the need to address platform integration during the integration effort unnecessary. Finally, the inherent structure of *AFIT*tool establishes a software process, making it unnecessary for the integration methodology to address process integration.

Both data and control integration were addressed in the design space defined for software tool integration. Functional dimensions, structural dimensions, and design rules were identified as part of the design space, following the description of a design space given by Lane [25]. Lane's original concept was extended to include two sets of functional dimensions, one set for a single tool and one set for a tool pair. The functional dimensions for a single tool provide a method of characterizing each tool, while the functional dimensions for the tool pair characterize the interface of the tools. The functional dimensions are applied successively, first for each tool and then for the tool pair, in order to reduce the dimensionality of the resulting design space. Together, the two sets of functional dimensions fully define a pair of tools to be integrated, while the structural dimensions define the resulting integrated system. Design rules provide a mapping from the functional dimensions to the structural dimensions for a given pair of tools.

The methodology developed as part of this research offers a step by step approach to tool integration. The tool integrator must first determine that there is information to be shared between the two tools. This information need not be in the same format, but must be semantically meaningful to both tools. Next, the methodology assists the integrator in classifying each tool in the pair with respect to its input mechanism, output mechanism, and extendability. The tool pair is also characterized with respect to its

extendability class and data compatibility. From this tool classification, a mapping is provided from the tool pair to the integrated system by applying the appropriate design rules. The resultant system is characterized by the communication path used, the data transformation mechanism, and control integration method.

This methodology was used to perform the tool integrations with *AFIT*tool. *AFIT*tool was integrated with three tools, the Acme parser, daVinci, and Rational Rose 98 [12] [10] [21]. The Acme parser is a tool that reads files written in the Acme architectural definition language and checks them for the proper syntax. After determining the syntax is correct, the file is reformatted for proper tabbing, as defined by the parser, and written to the output device. As part of the integration effort, the parser was extended slightly to allow input from stdin or a file and output to stdout or a file. In the integration with *AFIT*tool, a file was produced from the domain model using the Acme language to represent the event flows between classes in the model. The Acme parser is then called from *AFIT*tool and produces an output file, if the input is syntactically correct, in the proper style, and with the name supplied by the user.

The graph layout tool daVinci was used in the integration with *AFIT*tool to produce a data flow diagram (DFD) for the currently loaded domain model. In order to integrate the two tools, *AFIT*tool was extended to create a file in the format expected by daVinci and then to execute daVinci, supplying the filename as an input parameter. One DFD for each class in the domain is displayed by daVinci and can be closed when the user is finished with it.

Rational Rose 98 was integrated with *AFIT*tool to provide two types of functionality: allowing a user to specify a domain for *AFIT*tool through diagrams with the Unified Modeling Notation (UML) and allowing a user to generate Rose diagrams from an existing *Z* LaTeX file in *AFIT*tool format. The integration of these two tools provides the user with a semi-formal view of the domain, rather than just the formal notation that was available. Both Rose and *AFIT*tool were extended, although *AFIT*tool was only extended to allow a user to execute Rose from within *AFIT*tool. The majority of the extension was performed by using the Rose scripting language, Summit BasicScript, to achieve the transformations from diagrams to *Z* LaTeX and *Z* LaTeX to diagrams [21]. This integration

Table 10 Methodology for Tool Integration

	Rose Group	Non-Rose Group	Overall
Homework 2	562.6 min	677.9 min	620.2 min
Homework 3	167 min	230.8 min	194.4 min
Homework 5a	52.4 min	54.5 min	53.3 min
Project	36.2 hrs	60 hrs	46.3 hrs

also demonstrated that integrating two tools with a graphical user interface (GUI) lessens the ease of making the integration appear seamless. Although the two tools function well together, it is apparent that there are two separate tools executing.

5.2 Analyzing the Rose 98 Extensions

In order to gauge the merit of the integration of *AFIT*tool and Rational Rose, a small experiment was conducted between two groups of students. One group used the Rose extensions to produce the *Z* LaTeX files for *AFIT*tool, while the other group produced the files from a template with only a text editor. Students were assigned three homework problems and one project in which they were required to produce the *Z* LaTeX files. The homework problems were accomplished individually by 14 students and the project was accomplished by seven teams of two students each, including four teams using Rose. Although the group of students is not large enough to provide statistically valid results, the amount of time spent on each problem was collected and the results were examined for any trends that might indicate whether or not the Rose extensions are helpful.

These results, summarized in Table 10, display a trend indicating Rose users had consistently faster times than those who did not use Rose. While neither group of students is large enough to make statistically valid results, the consistency of these results clearly indicates that the extensions to Rose are helpful in reducing development time.

5.3 Conclusions

The integration efforts involving *AFIT*tool cover a representative sample of the categories discussed in the methodology. The methodology assists a tool integrator by first allowing him to characterize the pair of tools and then giving him a step by step approach

to accomplish the integration. For each integration attempted as part of this research, the methodology was effective at directing the proper steps to take for integration. Although there is more than one way to accomplish most integrations, the methodology offered a solution that achieved the goal in each case.

The integrations performed as part of this research effort serve to address two of *AFIT*tool's shortcomings, described in Chapter 2. Allowing the user to develop a domain model in Rose improves the user interface to *AFIT*tool, both by offering a graphical input method and by providing a faster method of input. Although the only accepted input to *AFIT*tool is the *Z* LaTeX file, now the user has the option of developing the file through Rose. Through the experiment performed to judge the value of this approach, it seems that giving the user the ability to use Rose to develop the domain model significantly decreases development time in some cases. The integrations have also made it possible for the user to perform model analysis on three aspects of the model: the DFD displayed by daVinci, the object model displayed by Rose, and the event flows written in Acme.

One of the initial goals of this research was to provide a step by step approach to a *seamless* integration. However, in the integration performed between *AFIT*tool and Rose, it is apparent that two tools were made to work together because Rose is an interactive tool with a GUI. Although daVinci also has a GUI, the integration of *AFIT*tool and daVinci gives a different appearance because the user does not interact with daVinci, except to close the windows. There are frameworks on the market that integrate tools with an existing GUI to make the integrated system look like one tool. It is possible that is the approach that should be taken when integrating two tools with a GUI. The methodology does not specifically address presentation integration, an important class of integration when tools with a GUI are involved. Developing a methodology that addresses presentation integration was outside the scope of this research, primarily due to time constraints.

5.4 Recommendations For Future Work

While this research effort accomplished quite a bit in the direction of a step by step approach to tool integration and also in improving *AFIT*tool through integration with

other tools, there are several possible areas for future researchers to pursue. The following sections describe possibilities for further work on both the methodology and *AFIT*tool.

5.4.1 Extending Methodology. Extending the methodology could be accomplished in several ways, one of which would be to consider the other three classes of integration: presentation, platform and process. In this effort, they were irrelevant and the methodology does not address them. As the area of integrating two tools with a GUI is further explored, it may be necessary to include a GUI as an alternative to the functional dimension input mechanism and to define design rules to address the extended dimension. The addition of GUI as an input mechanism may also make it necessary to consider presentation integration as part of the methodology since presentation integration addresses how the system looks to the user.

Platform and process integration should also be considered as extensions to the methodology. Accomplishing a cross-platform, multiple GUI tool integration would test areas of the methodology that were not considered in this effort and may point out areas that are lacking. By exploring platform integration, the integration methodology can also be used for distributed applications, which may or may not include different platforms. Adding these integration classes may require the addition of one or more functional dimensions for each integration class and design rules to map from the functional to the structural dimensions. Additionally, it may be necessary to add one or more structural dimensions in order to characterize the resulting system with the additional consideration of presentation, process and/or platform.

5.4.2 Extending Existing AFIT*tool Interface.* The integrations accomplished between *AFIT*tool and Rose can also be improved. Since the primary goal was to demonstrate integration of the tools, the extensions to Rose are not as complete as they could be. Neither one of the Rose scripts handles cardinality of associations to the fullest. When supplied with a LaTeX file, the Rose script simply alerts the user to place the proper cardinality on the association. Since Rose treats aggregation as a special case of an association, this is true for aggregation also. The user is required to enter the cardinality of the aggregate components. When Rose writes the LaTeX file for *AFIT*tool, it does not write all of

the constraints due to the association. It does, however, output the proper constraints for aggregation.

5.4.3 Further AFIT*tool Integration.* Regarding improving *AFIT*tool, an untapped area is that of developing a state model simulator. It would be helpful for the user to be able to see their state model "in action." During this research, the only tools found that are capable of this type of simulation would have required a large number of changes before integration. A more extensive search may locate a more appropriate tool. Alternatively, a simulator could be developed at AFIT and integrated with *AFIT*tool. The layout portion of the simulator could be accomplished by integrating with daVinci through the remote procedure call (RPC) interface. With the layout accomplished, the task would involve developing a method of testing all possible routes through the state model, based on the information in the state transition table.

Integrating a theorem prover with *AFIT*tool would also be beneficial since *AFIT*tool is based on formal methods. There are several theorem provers available, some free and some for purchase, that would enable an *AFIT*tool user to check the correctness of the specification entered into the domain model. Additionally, there are theorem provers that could examine the code produced by *AFIT*tool and check it for errors. The integration of these tools, however, is only part of the effort. The larger effort would be determining what information to provide as input to the theorem prover, i.e., determining what entails a correct specification or correct code.

5.5 Summary

The main contribution of this thesis effort is the development of a step by step approach to tool integration. By following the approach described here, tools can be integrated to improve an existing tool and/or provide a unified view of several tools. By choosing tools based on their characteristics and their ability to address the shortcomings of *AFIT*tool, the methodology was demonstrated and two of the shortcomings identified for *AFIT*tool have been addressed through tool integration. The methodology developed

is capable of guiding the tool integrator to a feasible solution and can be applied to the integration of any two software tools.

Appendix A. AFIT*tool Input Template*

ObjectX Structure Definition

Object Name: ObjectX

Object Number: 9404XX

Object Description:

Date: 10/01/96

History:

Author: Hartrum

Superclass: None

Components: None

Context: None

Attributes:

Constraints:

None

Z Static Schema:

Let **SSAN** be the set of all Social Security numbers.
Let **DATE** be the set of all calander dates.
Let **GENDER** be the set of gender types.

$[SSAN, DATE]$

$GENDER ::= male \mid female$

ObjectX

*attribute*1 : *type* *attribute*2 : *type*
*predicate*1 *predicate*2

$$
\begin{array}{|l}
\hline
InitObjectX \\
\Delta ObjectX \\
\hline
attribute1' = value1 \\
attribute2' = value2 \\
\hline
\end{array}
$$

AssocW Association Definition

Association Name: AssocW

Association Number: 9404XX

Association Description:

Date: 10/03/94

History:

Author: Hartrum

First Object Class: ObjectX

Multiplicity:

Second Object Class: ObjectY

Multiplicity:

Context: None

Attributes:

None

Constraints:

None

Z Static Schema:

Let **SSAN** be the set of all Social Security numbers.

AssocAttr

$attribute1 : type$
$attribute2 : type$

$predicate1$
$predicate2$

AssocW

$assocw : ((ObjectX \times ObjectY) \rightarrow AssocAttr)$

$predicate1$
$predicate2$

ObjectX Functional Model

Object: *ObjectX*

Process Name:

Process Description:

Z Dynamic Schema:

ProcessName

$\Delta ObjectX$
$input1? : type$
$output1! : type$
$localvar1 : type$

---

preconditions
postconditions

ObjectX Dynamic Model

State Name:

State Description:

Z Static Schema:

StateA

$$ObjectX$$

$$attributeA > 0$$
$$attributeA < attributeB$$

Event Name:

Event Description:

Z Static Schema:

EventA

$$parameter_1 : TYPE$$
$$parameter_2 : TYPE$$

$$parameter_1 > 0$$

State Transition Table:

Current	Event	Guard	Next	Action	Send
StateA	Event1	$a < b$	StateB	Action-A	EventA

Appendix B. Z Symbols

α	\alpha	θ	\theta	o	o	τ	\tau
β	\beta	ϑ	\vartheta	π	\pi	υ	\upsilon
γ	\gamma	γ	\gamma	ϖ	\varpi	ϕ	\phi
δ	\delta	κ	\kappa	ρ	\rho	φ	\varphi
ϵ	\epsilon	λ	\lambda	ϱ	\varrho	χ	\chi
ε	\varepsilon	μ	\mu	σ	\sigma	ψ	\psi
ζ	\zeta	ν	\nu	ς	\varsigma	ω	\omega
η	\eta	ξ	\xi				

	\Gamma	Λ	\Lambda	Σ	\Sigma	Ψ	\Psi
Δ	\Delta	Ξ	\Xi	Υ	\Upsilon	Ω	\Omega
Θ	\Theta	Π	\Pi	Φ	\Phi		

Table 1: Greek Letters

$\pm$	\pm	$\cap$	\cap	$\diamond$	\diamond	$\oplus$	\oplus
$\mp$	\mp	$\cup$	\cup	$\triangle$	\bigtriangleup	$\ominus$	\ominus
$\times$	\times	$\uplus$	\uplus	$\triangledown$	\bigtriangledown	$\otimes$	\otimes
$\div$	\div	$\sqcap$	\sqcap	$\triangleleft$	\triangleleft	$\oslash$	\oslash
$*$	\ast	$\sqcup$	\sqcup	$\triangleright$	\triangleright	$\odot$	\odot
$\star$	\star	$\vee$	\vee	$\lhd^b$	\lhd	$\bigcirc$	\bigcirc
$\circ$	\circ	$\wedge$	\wedge	$\rhd^b$	\rhd	$\dagger$	\dagger
$\bullet$	\bullet	$\setminus$	\setminus	$\unlhd^b$	\unlhd	$\ddagger$	\ddagger
$\cdot$	\cdot	$\wr$	\wr	$\unrhd^b$	\unrhd	$\amalg$	\amalg
$+$	+	$-$	-				

[b] Not predefined in a format based on `basefont.tex`. Use one of the style options `oldlfont`, `newlfont`, `amsfonts` or `amssymb`.

Table 2: Binary Operation Symbols

$\leq$	\leq	$\geq$	\geq	$\equiv$	\equiv	$\models$	\models
$\prec$	\prec	$\succ$	\succ	$\sim$	\sim	$\perp$	\perp
$\preceq$	\preceq	$\succeq$	\succeq	$\simeq$	\simeq	$\mid$	\mid
$\ll$	\ll	$\gg$	\gg	$\asymp$	\asymp	$\parallel$	\parallel
$\subset$	\subset	$\supset$	\supset	$\approx$	\approx	$\bowtie$	\bowtie
$\subseteq$	\subseteq	$\supseteq$	\supseteq	$\cong$	\cong	$\Join^b$	\Join
$\sqsubset$	\sqsubset^b	$\sqsupset$	\sqsupset^b	$\neq$	\neq	$\smile$	\smile
$\sqsubseteq$	\sqsubseteq	$\sqsupseteq$	\sqsupseteq	$\doteq$	\doteq	$\frown$	\frown
$\in$	\in	$\ni$	\ni	$\propto$	\propto	$=$	=
$\vdash$	\vdash	$\dashv$	\dashv	$<$	<	$>$	>
$:$	:						

[b] Not predefined in a format based on `basefont.tex`. Use one of the style options `oldlfont`, `newlfont`, `amsfonts` or `amssymb`.

Table 3: Relation Symbols

,	,	;	;	:	\colon	.	\ldotp	·	\cdotp

Table 4: Punctuation Symbols

$\leftarrow$	\leftarrow	$\longleftarrow$	\longleftarrow	$\uparrow$	\uparrow
$\Leftarrow$	\Leftarrow	$\Longleftarrow$	\Longleftarrow	$\Uparrow$	\Uparrow
$\rightarrow$	\rightarrow	$\longrightarrow$	\longrightarrow	$\downarrow$	\downarrow
$\Rightarrow$	\Rightarrow	$\Longrightarrow$	\Longrightarrow	$\Downarrow$	\Downarrow
$\leftrightarrow$	\leftrightarrow	$\longleftrightarrow$	\longleftrightarrow	$\updownarrow$	\updownarrow
$\Leftrightarrow$	\Leftrightarrow	$\Longleftrightarrow$	\Longleftrightarrow	$\Updownarrow$	\Updownarrow
$\mapsto$	\mapsto	$\longmapsto$	\longmapsto	$\nearrow$	\nearrow
$\hookleftarrow$	\hookleftarrow	$\hookrightarrow$	\hookrightarrow	$\searrow$	\searrow
$\leftharpoonup$	\leftharpoonup	$\rightharpoonup$	\rightharpoonup	$\swarrow$	\swarrow
$\leftharpoondown$	\leftharpoondown	$\rightharpoondown$	\rightharpoondown	$\nwarrow$	\nwarrow
$\rightleftharpoons$	\rightleftharpoons	$\leadsto$[b]	\leadsto[b]		

[b] Not predefined in a format based on `basefont.tex`. Use one of the style options `oldlfont`, `newlfont`, `amsfonts` or `amssymb`.

Table 5: Arrow Symbols

$\ldots$	\ldots	$\cdots$	\cdots	$\vdots$	\vdots	$\ddots$	\ddots
$\aleph$	\aleph	$\prime$	\prime	$\forall$	\forall	∞	\infty
$\hbar$	\hbar	$\emptyset$	\emptyset	$\exists$	\exists	$\Box$	\Box[b]
$\imath$	\imath	∇	\nabla	$\neg$	\neg	$\Diamond$	\Diamond[b]
$\jmath$	\jmath	$\surd$	\surd	$\flat$	\flat	$\triangle$	\triangle
ℓ	\ell	$\top$	\top	$\natural$	\natural	$\clubsuit$	\clubsuit
$\wp$	\wp	$\bot$	\bot	$\sharp$	\sharp	$\diamondsuit$	\diamondsuit
$\Re$	\Re	$\|$	|	$\backslash$	\backslash	$\heartsuit$	\heartsuit
$\Im$	\Im	$\angle$	\angle	∂	\partial	$\spadesuit$	\spadesuit
$\mho$	\mho[b]	.	.	$\|$	|		

[b] Not predefined in a format based on `basefont.tex`. Use one of the style options `oldlfont`, `newlfont`, `amsfonts` or `amssymb`.

Table 6: Miscellaneous Symbols

$\sum$	\sum	$\bigcap$	\bigcap	$\bigodot$	\bigodot
$\prod$	\prod	$\bigcup$	\bigcup	$\bigotimes$	\bigotimes
$\coprod$	\coprod	$\bigsqcup$	\bigsqcup	$\bigoplus$	\bigoplus
$\int$	\int	$\bigvee$	\bigvee	$\biguplus$	\biguplus
$\oint$	\oint	$\bigwedge$	\bigwedge		

Table 7: Variable-sized Symbols

\arccos	\cos	\csc	\exp	\ker	\limsup	\min	\sinh
\arcsin	\cosh	\deg	\gcd	\lg	\ln	\Pr	\sup
\arctan	\cot	\det	\hom	\lim	\log	\sec	\tan
\arg	\coth	\dim	\inf	\liminf	\max	\sin	\tanh

Table 8: Log-like Symbols

(	(	)	)	$\uparrow$	\uparrow	$\Uparrow$	\Uparrow
[	[	]	]	$\downarrow$	\downarrow	$\Downarrow$	\Downarrow
{	\{	}	\}	$\updownarrow$	\updownarrow	$\Updownarrow$	\Updownarrow
$\lfloor$	\lfloor	$\rfloor$	\rfloor	$\lceil$	\lceil	$\rceil$	\rceil
$\langle$	\langle	$\rangle$	\rangle	/	/	$\backslash$	\backslash
$\|$	|	$\|$	|				

Table 9: Delimiters

$\rmoustache$	`\rmoustache`	$\lmoustache$	`\lmoustache`	$\rgroup$	`\rgroup`	$\lgroup$	`\lgroup`
$\arrowvert$	`\arrowvert`	$\Arrowvert$	`\Arrowvert`	$\bracevert$	`\bracevert`		

Table 10: Large Delimiters

$\hat{a}$	`\hat{a}`	$\acute{a}$	`\acute{a}`	$\bar{a}$	`\bar{a}`	$\dot{a}$	`\dot{a}`	$\breve{a}$	`\breve{a}`
$\check{a}$	`\check{a}`	$\grave{a}$	`\grave{a}`	$\vec{a}$	`\vec{a}`	$\ddot{a}$	`\ddot{a}`	$\tilde{a}$	`\tilde{a}`

Table 11: Math mode accents

$\widetilde{abc}$	`\widetilde{abc}`	$\widehat{abc}$	`\widehat{abc}`
$\overleftarrow{abc}$	`\overleftarrow{abc}`	$\overrightarrow{abc}$	`\overrightarrow{abc}`
$\overline{abc}$	`\overline{abc}`	$\underline{abc}$	`\underline{abc}`
$\overbrace{abc}$	`\overbrace{abc}`	$\underbrace{abc}$	`\underbrace{abc}`
$\sqrt{abc}$	`\sqrt{abc}`	$\sqrt[n]{abc}$	`\sqrt[n]{abc}`
f'	`f'`	$\frac{abc}{xyz}$	`\frac{abc}{xyz}`

Table 12: Some other constructions

In order to transform the diagram correctly, rules on the specification process have been established. These rules are described in the following sections.

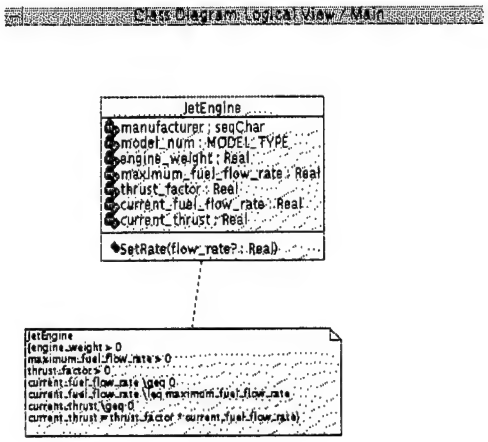


Figure 11 Class Diagram

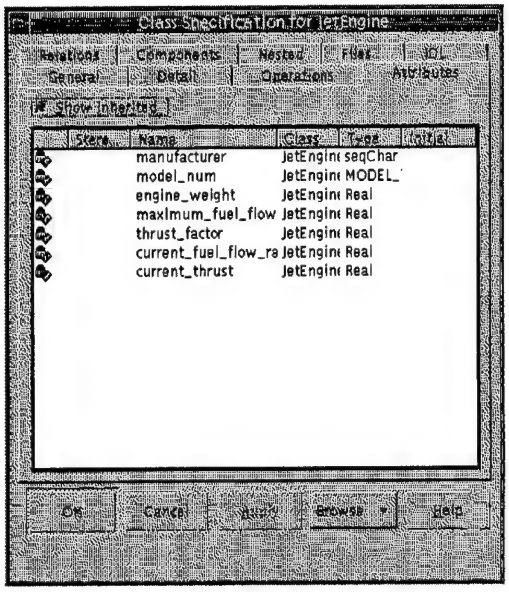


Figure 12 Attribute Declarations in Class

C.0.1 The Class Diagram. Each aggregate and primitive class must be fully specified, including its name, description, class constraints, attributes, types, and operations, including pre-conditions and post-conditions. An illustration of the class diagram is in Figure 11. If the domain model includes one or more aggregate classes, they must be included in the diagram. The lines from the aggregate to the component classes must be present as well. The transformation process will create one attribute for each component class, named *class name*Attr. If the aggregate lines include a name, it is ignored by the transformation process. The class name must be one word, but may include underscores. The class description is placed in the Documentation field of the class specification and may be several lines. Class constraints are placed in a Note box connected to the class by a

dotted line. The Class name must be the first thing in the note, and the only thing on the first line. The rest of the constraint is enclosed in braces ({ *constraint* }). Each statement in the constraint must be on a separate line. Any math symbols must be specified in the LaTeX manner. A full specification of these symbols can be found in Appendix B of this document. An example of a class constraint is as follows:

```
JetPropulsionSystem
{fuel_level \leq fuel_max
fuel_flow > 0}
```

Attributes must be defined in the class specification, as illustrated in Figure 12. Attribute and type names must be one word, but can include underscores. Types may be user-defined or system types. If an enumerated type is needed, the user may define this in the type field of the attribute specification as follows:

```
{nuclear | air-to-air | surface-to-air}
```

The type will be named during transformation using the attribute name with "Type" appended to it. If the above type definition was for the attribute "Missile", the type would be named "MissileType" during the transformation. It would be defined in Z as expected by *AFIT*tool.

Rose offers a section of the class specification for operation definitions. For a correct transformation, the definition of each operation must include the name, input parameters, output parameters, a description of the operation, pre-conditions and post-conditions of the operation. The operation name and parameters must be one word, but may include underscores, illustrated in Figure 13. The operation description is written in the documentation field of the operation specification. It may be multiple lines or it may be left blank. Each input parameter is specified in the parameter section of the specification. The same rules for naming and type definitions apply here as in the attribute section, with one exception: each input parameter name must end in a "?" due to constraints during generation of the LaTeX file. Output parameters are specified in the space for Return Type in the operation specification and must end in "!". Although Rose does not expect this field to contain more than one value, or names for the values, *AFIT*tool expects both names and types for output parameters of an operation. For this reason, the Return Type

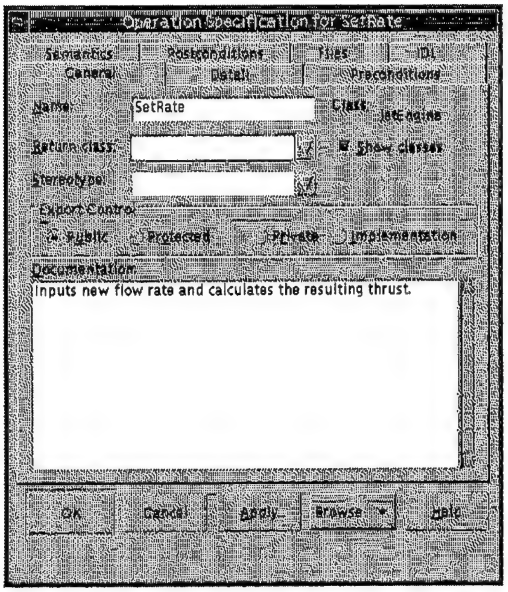


Figure 13 Operation Specification

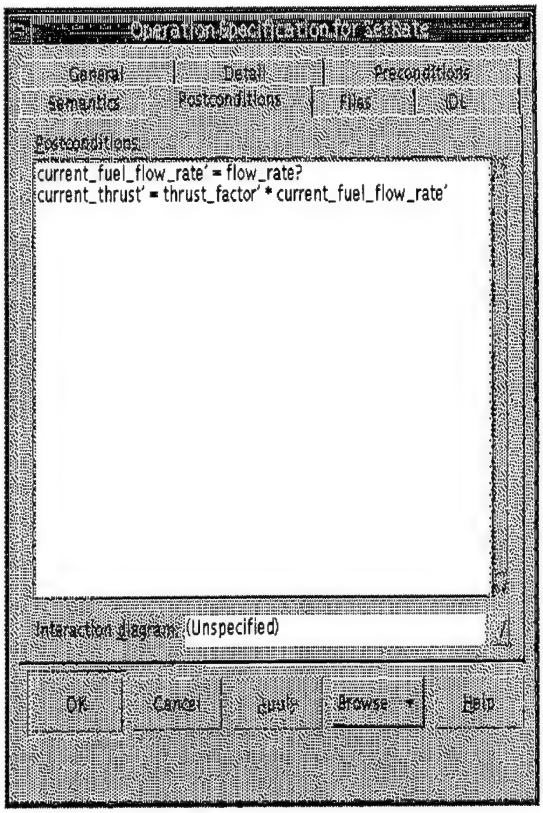


Figure 14 Post-Conditions in Operation

field must contain **ReturnParamName! : Type**. If there are multiple return parameters, the syntax is as follows: **ReturnParamName! : Type, ReturnParamName! : Type**. Pre-conditions and post-conditions are also specified for each operation. For an operation to be valid, it must have at least a post-condition. The attribute names used must be the same as those specified in the class definition. If an attribute is being changed, it must be followed by a single tick to indicate its post-operation value. Each pre-condition or post-condition must be stated on a separate line in Rose, illustrated in Figure 14. An example of a post-condition follows.

```
current_fuel_flow_rate' = flow_rate?
current_thrust' = thrust_factor' * current_fuel_flow_rate'
```

Associations between classes are illustrated in Rose by a solid line. They are only transformed and included in the domain model if they occur between two classes that are components of an aggregate class, also present in the Rose diagram. The domain and/or

range restrictions that may be necessary to fully specify an association must be entered in the class constraints for the aggregate class, as the Rose script does not generate those. This includes the cardinality on either end of the association. If the association involves an associative object, it should be present in the Rose diagram as well. It will be transformed and included as part of the aggregate class. The association will be named based on the name that is provided in the class hanging from the association line. Although Rose allows the name of the association to be entered on the line or in the class, the transformation expects the name in the class.

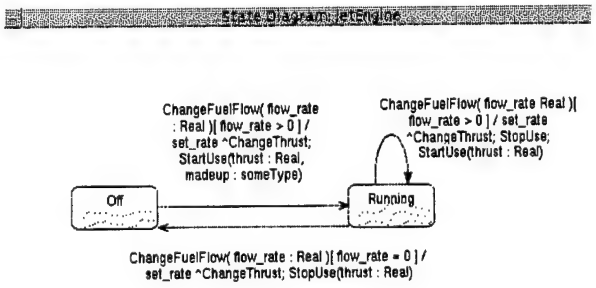


Figure 15 State Diagram

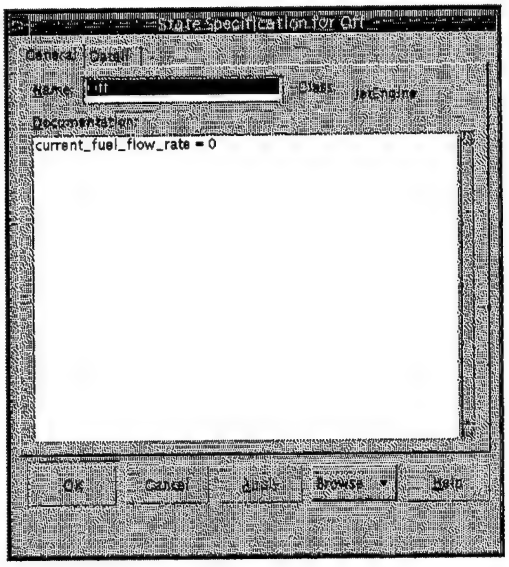


Figure 16 State Specification

C.0.2 The State Model. The state model in Rose is associated with one class. An example of a state diagram is illustrated in Figure 15. It contains states and transitions between the states. Each state must be fully specified, including the name and constraints that hold during that state. Rose does not offer a place for state constraints, so they are placed in the Documentation region of the state specification, listing one constraint per

line, illustrated in Figure 16. The rules for state constraints are the same as those for class constraints, with regard to using math symbols.

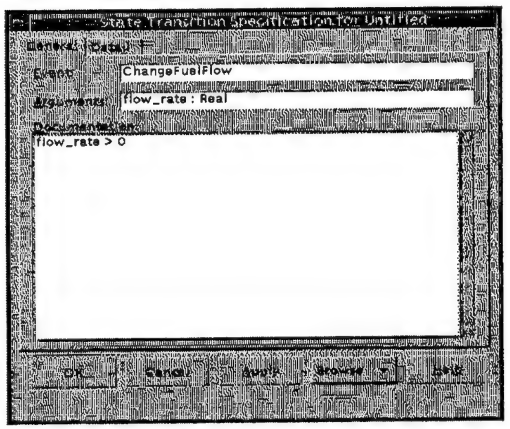


Figure 17 Transition Specification

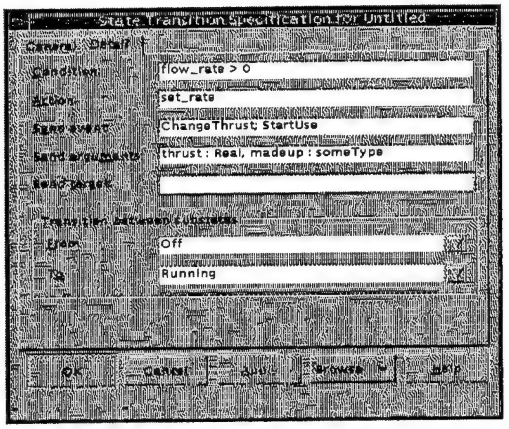


Figure 18 Detailed Transition Specification

The transition may include a trigger event, a guard condition on that trigger event, one or more send events and/or one or more actions to perform during the transition. The trigger event is named in the field for Event in the Transition Specification, illustrated in Figure 17. Any parameters to this event are placed in the Arguments field, with the name of the parameter and its type, as follows: `ParameterName : Type`. If an event has multiple parameters, they are listed as follows: `ParameterName : Type, ParameterName : Type`. Any constraints on the event parameters are placed in the Transition Documentation field, with the same format as previously described.

Actions are specified in the Action field of the Transition Specification. Parameters to the action are not specified. Each action should be defined in the Operations section of the Class definition. The guard condition is placed in the Condition field, using the math syntax as described earlier. The specification of Send Events is placed in the fields called Send Actions and Send Action Arguments in the Rose Transition Specification. If there are multiple Send Events, they are separated by semi-colons. Arguments for these events are also separated by semi-colons, leaving a space if one of the events has no arguments. If

one send event has multiple arguments, the arguments should be separated by commas, as described above for event parameters. The previous guidelines are illustrated in Figure 18. A textual example of three send events with only two having parameters is below. Notice the third event has two parameters, separated by a comma.

```
Send Action:  ChangeThrust; StopFlow; Schedule
Send Action Arguments:  thrust : Real; ; s_time : Real, s_priority : Nat
```

Appendix D. Detailed Descriptions of Design Rules

D.1 First Extend

- If stdout/stdin, it is preferred to extend the first tool in sequence to convert the data and then develop a driver program to execute the first tool followed by the second tool, with output from the first converted before it is output via stdout. This is an exception to the rule of using distributed control. In this case, centralized control should be used. This approach is chosen because the interface between the tools lends itself to a simple driver program to handle the integration.
- If stdout/file, it is preferred to extend the first tool to perform data integration followed by executing the second tool. This approach is preferred due to the properties of stdout, including the ease of capturing the data and writing it to a properly formatted file.
- If file/stdin, due to the characteristics of stdin, it is preferred to extend the first tool to build a command that performs any necessary data conversions, redirects stdin to data from a file, and executes the second tool.
- If file/file, it is preferred to extend the first tool to perform any necessary data conversions before executing the second tool. Since both tools use the same data medium, files, the least complex approach is to simply prepare the file for the second tool and execute that tool.
- If (anything)/message passing, it is preferred to require the first tool to gather the data needed by the second tool and format it for the messages before executing the second tool. While the second tool is executing, the first tool passes it the messages it expects, at the proper time and in the proper format, eliminating the need to change the second tool in any way.
- If message passing/(anything), except message passing/message passing, it is preferred to extend the first tool to capture all of the messages in a file and integrate like file/(anything). Since files are easily manipulated and read, the best approach is to capture the messages in a file.

- If built-in output/stdin, it is preferred to extend the first tool to capture the output and send it to stdin, since that's what the second tool expects, in the proper format before executing the second tool.

D.2 Second Extend

- If stdout/stdin, it is preferred to develop a separate program to perform any necessary data conversions. Additionally, develop a driver program to execute the first tool, followed by executing the data converter and sending that output to the second tool. This is an exception to the rule of using distributed control. In this case, centralized control will be used. This approach is taken due to the ease with which stdout and stdin can be manipulated.
- If stdout/file, it is preferred to extend the second tool to execute the first tool, save the output in a file, perform data integration and execute the necessary functions in the second tool. This approach is preferred due to the properties of stdout, including the ease of capturing the data and writing it to a properly formatted file.
- If (anything)/message passing, it is preferred to perform the same type of integration as with *First Extend* with the second tool extended rather than the first (if applicable) since the integrations are similar. Having the data ready for message passing smoothes the integration and allows successful integration without changing the first tool.
- If file/stdin or file/file, it is preferred to extend the second tool to execute the first tool, perform any necessary data conversions and then execute its own functions. The manipulations possible with files allow for any conversion and redirection necessary.
- If message passing/(anything), it is preferred to approach this integration in the same manner as *First Extend* with the second tool extended rather than the first (if applicable). This approach is taken due to the similarity between *Second Extend* and *First Extend.*
- If built-in output/(anything), it is preferred to develop a driver program that executes the first tool, intercepts output to stderr or the printer and saves it to a file, due to

the ease of file manipulation. This solution will use centralized control. The rest of the integration can then be performed in the same manner as file/(anything).

Appendix E. Acme Example for Aggregate Class

The following Acme code provides an example for the aggregate class Jet Propulsion System. Notice each component, **FuelTank**, **JetEngine**, and **Throttle** each has its own section of code, describing the event flows into and out of that component. This example also illustrates the flow between components of an aggregate class.

The state transition tables are first, followed by the actual Acme code generated.

Current	Event	Guard	Next	Action	Send
Empty	StartFill		Filling	set_inflow	Schedule
PartiallyFilled	StartFill		Filling	set_inflow	Schedule
PartiallyFilled	StartUse		Using	set_outflow	Schedule
Full	StartFill		Full		Overflow
Full	StartUse		Using	set_outflow	Schedule
Filling	StartUse		FillAndUse	set_outflow_level	Cancel
Filling	StopFill	$fuel\_level < capacity$	PartiallyFilled	set_update_level	Cancel
Filling	StopFill	$fuel\_level = capacity$	Full	set_update_level	Cancel
Filling	TankFull		Full	update_level	Overflow
Using	TankEmpty		Empty	update_level	ChangeFuelFlow
Using	StopUse		PartiallyFilled	set_outflow_level	Cancel
Using	StartFill		FillAndUse	set_inflow_level	Cancel
FillAndUse	StopFill		Using	set_inflow_level	Schedule
FillAndUse	StopUse		Filling	set_outflow_level	Schedule

Table 11 Fuel Tank State Table

Current	Event	Guard	Next	Action	Send
Off	ChangeFuelFlow	$flow\_rate > 0$	Running	set_rate	ChangeThrust; StartUse
Running	ChangeFuelFlow	$flow\_rate > 0$	Running	set_rate	ChangeThrust; StopUse; StartUse
Running	ChangeFuelFlow	$flow\_rate = 0$	Off	set_rate	ChangeThrust; StopUse

Table 12 Jet Engine State Table

Current	Event	Guard	Next	Action	Send
Normal	ChangeSetting		Normal	update_position_index	ChangeFuelFlow

Table 13 Throttle State Table

```
Family ObjectEvent = {
  Port Type SendPort = {
  };
  Port Type ReceivePort = {
  };
  Role Type Source = {
  };
  Role Type Destination = {
  };
  Component Type AggregateClass = {
  };
  Component Type PrimitiveClass = {
  };
  Connector Type EventFlow = {
  };
    Attachments {
    };
};
System JPS : ObjectEvent = {
  Component JetPropulsionSystem : AggregateClass = {
    Port ChangeSetting : SendPort;
    Port ChangeThrottle : ReceivePort;
    Port OutOfFuel : ReceivePort;
    Port StartEngines : ReceivePort;
    Port TankEmpty : SendPort;
    Port startup : ReceivePort;
    Representation {
      System Aggregate-rep : ObjectEvent = {
        Component FuelTank : PrimitiveClass = {
          Port Cancel : SendPort;
          Port ChangeFuelFlow : SendPort;
          Port Overflow : SendPort;
          Port Schedule : SendPort;
          Port StartFill : ReceivePort;
          Port StartUse : ReceivePort;
          Port StopFill : ReceivePort;
          Port StopUse : ReceivePort;
          Port TankEmpty : ReceivePort;
          Port TankFull : ReceivePort;
        };
        Component JetEngine : PrimitiveClass = {
          Port ChangeFuelFlow : ReceivePort;
          Port ChangeThrust : SendPort;
          Port StartUse : SendPort;
          Port StopUse : SendPort;
```

```
};
Component Throttle : PrimitiveClass = {
  Port ChangeFuelFlow : SendPort;
  Port ChangeSetting : ReceivePort;
};
Connector CancelEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
};
Connector ChangeFuelFlowEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
};
Connector ChangeSettingEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
};
Connector ChangeThrustEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
};
Connector OverflowEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
};
Connector ScheduleEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
};
Connector StartFillEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
};
Connector StartUseEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
};
Connector StopFillEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
};
Connector StopUseEvent : EventFlow = {
  Role sink : Destination;
  Role source : Source;
};
```

```
        Connector TankEmptyEvent : EventFlow = {
          Role sink : Destination;
          Role source : Source;
        };
        Connector TankFullEvent : EventFlow = {
          Role sink : Destination;
          Role source : Source;
        };
        Attachments {
          FuelTank.Schedule to ScheduleEvent.source;
          FuelTank.Overflow to OverflowEvent.source;
          FuelTank.ChangeFuelFlow to ChangeFuelFlowEvent.source;
          FuelTank.Cancel to CancelEvent.source;
          FuelTank.StopUse to StopUseEvent.sink;
          FuelTank.TankEmpty to TankEmptyEvent.sink;
          FuelTank.TankFull to TankFullEvent.sink;
          FuelTank.StopFill to StopFillEvent.sink;
          FuelTank.StartUse to StartUseEvent.sink;
          FuelTank.StartFill to StartFillEvent.sink;
          JetEngine.ChangeThrust to ChangeThrustEvent.source;
          JetEngine.StopUse to StopUseEvent.source;
          JetEngine.StartUse to StartUseEvent.source;
          JetEngine.ChangeFuelFlow to ChangeFuelFlowEvent.sink;
          Throttle.ChangeFuelFlow to ChangeFuelFlowEvent.source;
          Throttle.ChangeSetting to ChangeSettingEvent.sink;
        };
      }; /* end system */
    }
  };
}; /* end system */
```

Appendix F. Configuration Management of Files Related to this Research

Several files were generated as part of the implementation of this thesis effort. The location and content of these files is given in the following sections.

F.1 AFIT*tool*

File Name	Content
runcmd.lisp	Runs a command in the command shell; used to invoke daVinci, Acme parser, and Rose
dom2acme.re	Refine code to extract information from domain model and produce Acme architecture file
at2dav.re	Refine code to extract information from domain model and produce daVinci diagrams
domtool.re	Extended menu to include options for daVinci, Acme, and Rose

In order to use these files with *AFIT*tool, they need to be included in the system.lisp file, or compiled and loaded individually after Refine is started. They need to be compiled and loaded in the order in which they appear in the table.

F.2 daVinci

File Name	Content
/apps/AI/bin/SUN4SOL2/daVinci	daVinci Executable

Nothing has to be compiled for the daVinci integration, except the aforementioned files that are compiled as part of *AFIT*tool.

F.3 Rose

File Name	Content
SCRIPT_PATH$/Rose2at.ebs	Extracts information from Rose diagram to create a *Z* LaTeX file for *AFIT*tool
SCRIPT_PATH$/ltx2rose.ebs	Parses LaTeX file to create Rose diagrams
On the Hawkeye System:	
/apps/roseada/releases/rose.4.5.8153/rose.mnu	Menu file extended for *AFIT*tool integrations
/apps/roseada/releases/rose.4.5.8153/bin/rose.exe	Rose executable
On the PC Network:	
r:\simulat.ion\rose98\rose.mnu	Menu file extended for *AFIT*tool integrations
r:\simulat.ion\rose98\bin\rose.exe	Rose executable

The script files used for the Rose to LaTeX and LaTeX to Rose conversions are in the proper directory for Rose to use them, defined by the environment variable SCRIPT_PATH$. SCRIPT_PATH$ is currently set to /apps/roseada/releases/rose.4.5.8153/scripts. The scripts are interpreted, rather than compiled, so nothing needs to be done to them in order for them to work correctly.

Bibliography

[1] Acker, Michael L. *An Examination of Multi-Tier Designs for Legacy Data Access.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, Dec 1997. AFIT/GCS/ENG/97D-01.

[2] Baker, Sean. *CORBA Distributed Objects Using Orbix.* ACM Press, 1997.

[3] Brown, N. and C. Kindel. "Distributed Component Object Model Protocol – DCOM/1.0." http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt.

[4] Chung, P. Emerald, et al. "DCOM and CORBA Side by Side, Step by Step, and Layer by Layer." http://www.bell-labs.com/ẽmerald/dcom_corba/Paper.html.

[5] Coglianese, Louis, et al. *Domain Analysis for the Avionics Domain Application Generation Environment of the Domain-Specific Software Architecture Project.* Technical Report ADAGE-IBM-92-11, Wright Laboratory Avionics Directorate: IBM Federal Systems Company, Nov 1993.

[6] Corporation, Microsoft, editor. *Microsoft Visual Basic Language Reference: Programming System for Windows.* One Microsoft Way: Microsoft Corporation, 1991.

[7] DeLoach, Scott A. *Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specification.* PhD dissertation, Air Force Institute of Technology, Wright-Patterson AFB, OH, June 1996. AFIT/DS/ENG/96-05, AD-A310 608.

[8] Faris, Chris, et al. *Knowledge-Based Software Assistant Advanced Development Model(KBSA/ADM).* Technical Report AFRL-IF-RS-TR-1998-194, Rome Research Site, Rome, New York: Air Force Research Laboratory, Information Directorate, Sep 1998.

[9] Fleming, R.T. and N. Wybolt. "CASE Tool Frameworks," *Unix Review*, *8*(12):24–32 (December 1990).

[10] Frohlich, Michael and Mattias Werner. *The Graph Visualization System daVinci — A User Interface for Applications.* Technical Report 5/94, University of Bremen, May 1994.

[11] Garlan, David. *An Introduction to the Aesop System.* Technical Report ARPA Grant F33615-93-1-1330, Pittsburgh, PA: Carnegie Mellon University, July 1995.

[12] Garlan, David, et al. "Acme: An Architecture Description Interchange Language." *Proceedings of CASCON'97.* 169–183. November 1997.

[13] Genesereth, Michael R. and Steven P. Ketchpel. "Software Agents," *Communications of the ACM*, 48–53 (July 1994).

[14] Hartrum, Thomas C. "Integrating Software Architecture Considerations into Software Transformation Systems." Unpublished, Oct 1997.

[15] http://dsse.ecs.soton.ac.uk/ chp/amn_proof/doc/index.html. "AMN-PROOF." Online Documentation.

[16] http://logic.stanford.edu/software/epilog. "An Overview of EPILOG 2.0 for LISP." Online Documentation.

[17] http://pavg.stanford.edu/rapide/overview.html. "Overview Of The Rapide Prototyping Project." Online Documentation, July 1997.

[18] http://www.cs.cmu.edu/afs/cs/project/able/www/wright/index.html. "The Wright Architecture Description Language." Online Documentation, July 1998.

[19] http://www.islandsoft.com. "Island Software Corp.." Online Documentation.

[20] http://www.lemma one.demon.co.uk/ProofPower. "ProofPower." Online Documentation.

[21] http://www.rational.com/products/rose/prodinfo.html. "Rational Rose 98 Product Information." Online Documentation.

[22] http://www.tomsawyer.com. "Graph Layout Toolkit." Online Documentation.

[23] Kissack, John. *Transforming Aggregate Object-Oriented Formal Specifications to Code*. MS thesis, AFIT/GCS/ENG/99M-09, Graduate School of Engineering, Air Force Institute of Technology (AU), 1999.

[24] Kromodimoeljo, Sentot, et al. *EVES Proof Checking and Browsing Final Report*. Technical Report FR-95-5482-06, Ottawa, Ontario, Canada: ORA Canada, December 1995.

[25] Lane, Thomas G. *Studying software architecture through design spaces and rules*. Technical Report CMU/SEI-90-TR-18 and ESD-90-TR-219 and CMU-CS-90-175, Carnegie Mellon University, Pittsburgh, PA: Software Engineering Institute, Nov 1990.

[26] Ousterhout, John K. "Tcl: An Embeddable Command Language." *Proceedings of Winter USENIX Conference*. 1990.

[27] Ousterhout, John K. "Scripting: Higher-Level Programming for the 21st Century," *IEEE Computer*, 23–30 (Mar 1998).

[28] Premerlani, William J. "An Object-Oriented Relational Database," *Communications of the ACM*, 99–109 (Nov 1990).

[29] "Rational Educational Grant, SEED Program." Rational Software Corporation, 18880 Homestead Road, Cupertino, CA 95014.

[30] Reasoning Systems Inc. *REFINE User's Guide*, 1990.

[31] Reiss, Levi and Joseph Radin. *X Window Inside and Out*. 2600 Tenth St.: Osborne McGraw-Hill, 1992.

[32] Rogers, Paul. "Object Database Management Systems," *OTA Off the Record Research* (Feb 1997).

[33] Tankersley, Travis W. *Generating Executable Code from Formal Specifications of Primitive Objects*. MS thesis, AFIT/GCS/ENG/99M-19, Graduate School of Engineering, Air Force Institute of Technology (AU), 1999.

[34] Thomas, Ian and Brian A. Nejmeh. "Definitions of Tool Integration for Environments," *IEEE Software*, 29–35 (Mar 1992).

[35] Wallnau, K.C. and P.H. Feiler. *Tool Integration and Environment Architectures*. Technical Report CMU/SEI-91-TR-11, Carnegie Mellon University, Pittsburgh, PA: Software Engineering Institute, May 1991.

[36] Wasserman, Anthony I. "Tool Integration in Software Engineering Environments." *Software Engineering Environments: Proc. Int'l Workshop on Environments*, edited by F. Long. 137–149. Springer-Verlag, Berlin, 1990.

Vita

Penelope Noe was born in New London, CT. After moving many times, living primarily in Alaska, Texas, and Maine, she graduated from Bangor High School in Bangor, ME in June 1991. She earned a B.A. in Computer Science at the University of Maine, Orono, Maine, and graduated on 13 May 1995. On the same day, she was commissioned in the Air Force after completing four years of Air Force ROTC on a four-year scholarship. Before coming to AFIT in August 1997, she spent two years at the Pentagon, working for the Single Agency Manager Communications Group.

Permanent address: 112 Beverly Drive
Fredericksburg, TX 78624